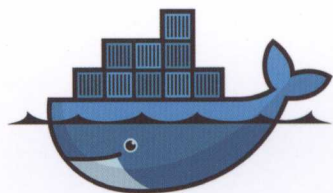


版权注意事项：1、书籍版权归著者和出版社所有；  
2、本PDF仅用于个人获取知识，进行私底下知识交流；  
3、PDF获得者不得在互联网以任何目的进行传播；  
如有需要，请尽量购买正版实体书！支持书籍作者！！



Docker不仅仅是颠覆程序部署的技术，  
更是软件开发中的一门艺术。



没什么难的

# Docker 入门与开发实战

Docker Entry and Development

熊昌隆◎编著

适合不同开发者的Docker工具书

- 内容全面：基础命令、进阶用法悉数囊括，知识点全覆盖。
- 活学活用：大量实践案例展示，指导上手使用，深化理解。
- 知识新鲜：以全新文档、材料为基础，跟进Docker演进的步伐。



中国工信出版集团



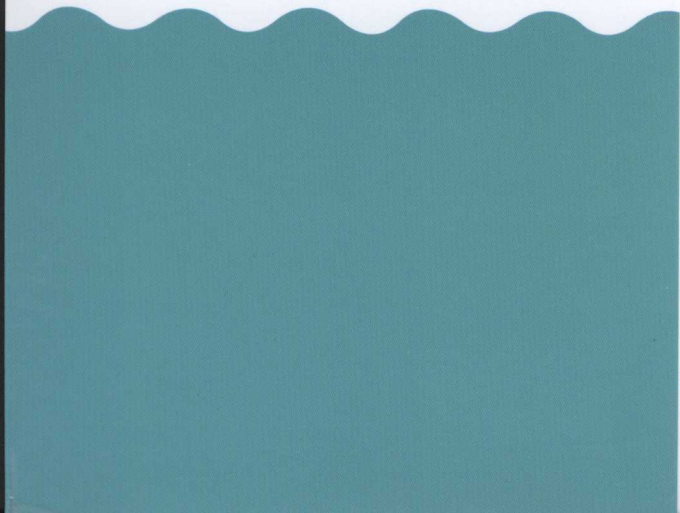
电子工业出版社  
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY  
<http://www.phei.com.cn>

---

## 熊昌隆:

全栈开发者, 技术布道师, 知名博主。  
参与智慧路由的研究, 多项研究成果影响了家用路由器的发展。设计实现的 Beaver 框架, 已经应用于千万级系统的程序架构中。发起组建的 Funcuter 开源小组, 旨在聚集国内的优秀开源项目和开发者, 目前已经拥有多个开源项目。一直致力于前沿技术的研究和推广, 希望让更多的开发者更轻松地接触、了解新兴科技成果, 并应用于开发实践中。

---



# 没什么难的

## 北京·BEIJING

## 内 容 简 介

作为引领近几年容器虚拟化领域的技术，Docker 的发展方兴未艾。但由于其出现不过数年，关于它的资料，特别是中文资料仍然相对匮乏，导致国内许多开发者对其感到陌生。本书正是以布道 Docker 为理念，由浅入深地从阐述 Docker 的基本概念、讲解常规使用方法、进行操作实践的演示、展示提高和进阶用法、剖析内部原理和底层架构等多个方面，全方面地展现 Docker 所具有的魅力。

本书由概念至实践，从不同方面向读者展现了 Docker，实用性非常强，既可以作为一本学习 Docker 的入门教材，也能作为进行 Docker 操作实践的说明书，甚至可以作为提高和进阶的知识宝库。不论是对 Docker 认知程度不同，有着不同知识储备的人，还是处于程序软件研发部署流程中的开发人员、测试人员或运维人员，本书都有着很强的可阅读性。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。  
版权所有，侵权必究。

## 图书在版编目 (CIP) 数据

没什么难的 Docker 入门与开发实战 / 熊昌隆编著. —北京：电子工业出版社，2017.6

ISBN 978-7-121-31427-8

I. ①没… II. ①熊… III. ①Linux 操作系统—程序设计 IV. ①TP316.85

中国版本图书馆 CIP 数据核字 (2017) 第 085024 号

责任编辑：李 冰

特约编辑：赵海红 罗树利等

印 刷：三河市华成印务有限公司

装 订：三河市华成印务有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱

邮编：100036

开 本：787×1092 1/16 印张：22.75 字数：582.4 千字

版 次：2017 年 6 月第 1 版

印 次：2017 年 6 月第 1 次印刷

定 价：69.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：(010) 88254888，88258888。

质量投诉请发邮件至 [zltz@phei.com.cn](mailto:zltz@phei.com.cn)，盗版侵权举报请发邮件至 [dbqq@phei.com.cn](mailto:dbqq@phei.com.cn)。

本书咨询联系方式：[libing@phei.com.cn](mailto:libing@phei.com.cn)。



# 前言

在快速发展的互联网领域，总在涌现引领潮流的新技术，最近几年，Docker 就成了这些技术中的一员。Docker 作为近几年备受关注的程序部署方案，实现了程序的快速部署，为分布式等场景下的部署提供了有力的帮助。在云计算及虚拟化领域，Docker 这个诞生不过数年的项目，只能算个新生儿，在功能完整性和稳定性方面，都不及其他已经经受过历史考验的项目。但为何仅仅几年的时光，就让 Docker 成为业界公认的优秀项目，关键在于 Docker 真正解决了分布式部署效率这一行业痛点。Docker 所提供的崭新分布式部署方案，不但像其他虚拟化方案一样，大幅减少了部署过程中适配环境所带来的额外工作，还充分弱化了虚拟化程序在虚拟化过程中对性能的影响，使得在 Docker 中运行的程序的效率能够与直接运行在真实操作系统中的程序的效率相媲美。

不过 Docker 能够受到各界追捧的原因并不仅仅在于其在部署领域带来的变化，其受到赞誉的原因也在于它能够打通开发、测试、运维等多个环节，为整个项目的开发流程营造统一的运行环境。由于 Docker 提供了非常轻量级的容器虚拟化方案，使得 Docker 能够以非常低的消耗运行在系统中。这也就使得我们不仅可以使用 Docker 在服务器中部署程序，也能在开发过程中利用 Docker 在本地系统中搭建程序运行环境。

由于 Docker 诞生不过数年，并且发展和迭代的速度非常快，所以其相关的教学资料比较匮乏，特别是中文文献，相对其他发展数十年的技术来说，简直是少之又少。本书正是建立在帮助希望了解和使用 Docker 的读者的基础上，收集了与 Docker 相关的资料，特别是缺少中文翻译的外文资料，集合整理成文，并与相关的案例、实践组合搭配，给读者提供了解 Docker 的捷径。

本书由浅及深，从不同维度解读和展现了 Docker 的概念、原理、使用方法、实践案例及周边工具，不同岗位的开发人员，或者对 Docker 有着不同认知的开发人员都能从中得到想要的知识。本书虽不能全面地阐述 Docker 的所有功能与特性，但系统性的知识梳理、理论与实践相结合的方式，都为读者了解和使用 Docker 提供了有效的帮助和指导。不论

你将本书看作 Docker 的入门教材，还是当成提升 Docker 知识储备的工具手册，都能为你带来不错的效果。

因受作者水平和成书时间所限，本书难免存有疏漏和不当之处，敬请指正。

## 本书特色

### 由浅及深，适合不同知识层面的读者

本书的内容涵盖了 Docker 的基础概念和常规使用方法，常见服务器程序在 Docker 中搭建和使用的实践，安全策略和辅助工具等知识概述，由浅入深，循序渐进，为不同的读者准备了不同的知识盛宴。对于 Docker 中的重点知识，必备、常用的操作方法和策略，本书不惜笔墨，进行了充分甚至反复的阐述和演示。而可供延展的知识点，虽然由于篇幅限制不能详细讲解，也都一一列出，供大家自行查阅，进行延伸阅读。本书的章节脉络清晰明确，使读者能循序渐进地掌握 Docker 知识，是一本不可多得的 Docker 资料手册和教材。

### 通俗易懂，理论与实践结合

本书的行文中穿插了很多对 Docker 使用方法的展示，并提供了专门的章节演示 Docker 的实践之道。通过这些操作示例的引导，避免了读者进行纸上谈兵式的阅读，也使得章节之间的知识可以由这些演示串联起来，能够减少知识脱节的现象发生。而对于理论知识的说明，本书绝不是生搬硬套地进行教条式的列举，也没有以堆砌专有词汇的方式简单概括，而是通过通俗语言将晦涩的知识以生活化的方式展现出来，让读者，特别是 Docker 初学者更容易地理解 Docker。

### 跟进时代，采取最新资料编写

Docker 是一门新兴技术，也是一门快速发展的技术，仅仅诞生数年就已经迭代了数十个版本。由于 Docker 在迭代的过程中不断地优化、完善、补充，所以不同版本之间所具有的功能和使用方法都存在很大的区别，因此学习 Docker 一定要使用新鲜的一手资料。本书在编写的过程中，收集和参考了大量最新的材料，特别是从 Docker 官方文档中提取了很多 Docker 最新的特性和使用方法，也从 Docker 的技术说明和源代码中总结了 Docker 的架构逻辑。由于本书是在对这些崭新的材料的收集汇总以及精心梳理的基础上完成的，所以本书可以为大家学习 Docker 提供强有力的支持。

## 本书的内容及体系结构

本书主要分为三部分，分别从基础、实践和提高的角度向读者介绍 Docker 的概念和使用方法。

第一部分为基础篇，包含了第 1~5 章的内容。在基础篇中，我们讲解了 Docker 的历史和基本概念，介绍最常见且最基础的 Docker 使用方法。

### 第 1 章 初始 Docker

本章从虚拟化、容器技术的发展历史与现状出发，逐步引入和展示 Docker 这项全新的虚拟化解决方案。除了向读者介绍 Docker 的组成结构及其发展历史，我们还将比较 Docker 与以往的部署及虚拟化方案的不同，分析使用 Docker 的优势所在，并介绍适合使用 Docker 的常见场景。另外，我们还将教会大家如何在常用的几种操作系统中安装 Docker。

### 第 2 章 镜像与仓库

本章由 Docker 镜像的概念出发，讲解 Docker 镜像的结构特点与组成形式，比较 Docker 镜像与其他虚拟化方案中镜像的异同，同时将常用的 Docker 镜像管理方法介绍给读者。除此之外，我们还将介绍如何使用 Docker 特有的镜像仓库存储、共享和迁移镜像，以及 Docker 官方所提供的 Docker Hub 镜像仓库的使用方法。

### 第 3 章 管理和使用容器

本章主要介绍了 Docker 的核心，也就是容器技术中容器的实现。本章将逐一谈及新增、运行、停止、删除等常用的容器操作方法，也将向读者展示如何查看容器的运行状态以及如何到容器中进行操作。另外，本章也会提及如何进行容器的迁移。

### 第 4 章 数据卷与网络

本章从容器的网络数据和文件数据的交换出发，引出和介绍了 Docker 提供的容器网络和数据卷这两个模块。在有关数据卷的部分，读者可以了解到数据卷的基本概念，以及如何创建或者从宿主机中挂载数据卷。在有关容器网络的部分，读者可以了解到容器网络的基本知识，以及如何让外部网络访问到容器或者实现容器间的网络通信。

### 第 5 章 制作镜像

本章主要介绍如何根据需要，通过编写 Dockerfile 构建 Docker 镜像。在展示了如何



将程序打包到镜像中之后，我们还对 Dockerfile 的写法以及可能使用到的指令进行了全方位的讲解。

第二部分是实践篇，由第 6~11 章的内容组成。在实践篇中，我们将以基础篇中所学到的知识为基础，分别对常见的服务器程序在 Docker 中的使用进行实践。

### 第 6 章 SSH 服务

本章主要展示了 SSH 服务在 Docker 容器中运行的方式，并介绍了 SSH 服务在 Docker 中所扮演的角色，还带领读者进行了在 Docker 容器中搭建 SSH 服务，以及构建包含 SSH 服务的 Docker 镜像的实践。

### 第 7 章 Web 服务器

本章首先简单介绍了 Web 服务和能够提供 Web 服务的常见程序，也分别在 Docker 容器中搭建了 Apache、Nginx 和 Tomcat 这几个常见的 Web 服务程序，还将通过编写 Dockerfile 的方式将这几款 Web 服务程序封装成 Docker 镜像。

### 第 8 章 数据库程序

本章对目前最受欢迎的开源关系型数据库 MySQL 和非关系型数据库 MongoDB 做简单介绍，展示如何在 Docker 容器中使用它们，也会提及如何让这些数据库向外提供服务。我们还将把这几款数据库软件通过 Dockerfile 构建成镜像，方便在 Docker 中使用。

### 第 9 章 缓存工具

本章首先提及在服务器中使用缓存工具的意义，并介绍 Memcached 和 Redis 这两款常用作处理缓存的工具。在了解 Memcached 和 Redis 的使用之后，我们还会将它们部署到 Docker 容器之中，并通过搭建程序的实践，将这两款程序封装到 Docker 镜像里。

### 第 10 章 动态处理程序

本章主要介绍了 Java、PHP、Python 和 Node.js 这几款常用的处理 Web 请求的程序，并讲解了如何在 Docker 容器中安装或搭建这些程序。在了解了这些程序在容器中安装或搭建的过程之后，我们还将通过 Dockerfile 将这几款软件封装成独立的 Docker 镜像。

### 第 11 章 综合演练

本章在之前所进行的实践的基础上，将实践过的 Web 服务程序、数据库程序、缓存



工具和动态处理程序，通过运行它们的 Docker 容器进行组合，讲解如何通过 Docker 构建和运行一套完整的 Web 服务体系。

第三部分是提高篇，汇总在第 12~16 章的内容中。在提高篇中，我们主要针对一些 Docker 更深入的使用方法和概念、原理进行学习和探究。

## 第 12 章 网络进阶

本章在之前所介绍的 Docker 网络基础概念和使用方法的基础之上，进一步深入地介绍了 Docker 网络的实现方法，阐述 Docker 网络的底层架构以及容器网络模型的概念，对用于管理容器网络的命令，也做了专门介绍。另外，本章还向读者介绍了如何进行深度定制的 Docker 容器网络配置和控制。

## 第 13 章 安全加固

本章从 Docker 的底层隔离机制出发，阐述了隔离机制是如何保证容器中程序互不干扰的，并由此展开，谈到了控制容器使用资源的方式和原理。我们还介绍了如何通过内核的安全机制及相关安全防护程序，来控制程序权限以及防范可能发生的攻击与破坏。另外，我们还将展示常用于 Docker 的安全策略和防护方法。

## 第 14 章 Docker API

本章主要讲解 Docker 中最基础也最重要的与外界沟通的方式，即 Docker API。在对 Docker API 的实现方式与分类进行介绍之后，我们还将抽取最常用的用于管理 Docker 核心模块的 Docker Remote API，以及用于与远程镜像仓库镜像交互的 Docker Registry API 进行专门的讲解和示范。

## 第 15 章 管理工具

本章主要展示了 Docker Compose、Docker Machine、Docker Swarm 这三个 Docker 周边的辅助程序，分别介绍了它们在各自领域辅助 Docker 的方式。同时，我们也分别介绍了它们的常见功能和基础使用方法。

## 第 16 章 Docker 的技术架构

本章对组成 Docker 核心的相关技术做了简单分析和介绍，包括提供容器技术支持的 Linux 内核命名空间和控制组，也囊括提供文件系统支持的 UFS。另外，本章还对 Docker 的核心程序包，也就是 Docker Engine 的组织架构进行了分析。通过对这些技术的简单介绍，可以为希望深入了解和探究 Docker 的读者提供一定的帮助。

## 本书的读者对象

- ❑ 希望了解 Docker 的初学者。
- ❑ 正在尝试使用 Docker 的程序员。
- ❑ 想要对自身所掌握的 Docker 知识加以提高的人。
- ❑ 有意将 Docker 引入项目中的架构师或管理人员。
- ❑ 需要在开发流程中使用 Docker 的研发、测试和运维的人。
- ❑ 其他对 Docker 感兴趣的开发者。

# 目 录

## 第一部分 基础篇

第 1 章 初识 Docker .....	1
1.1 虚拟化 .....	2
1.1.1 虚拟化技术 .....	2
1.1.2 虚拟化的分类 .....	4
1.2 容器技术与 Docker .....	6
1.2.1 容器技术 .....	6
1.2.2 Docker 简介 .....	8
1.2.3 改变世界的发明 .....	10
1.3 Docker 的安装 .....	10
1.3.1 在 Ubuntu 中安装 Docker .....	11
1.3.2 在 CentOS 中安装 Docker .....	12
1.3.3 在 Windows 中安装 Docker .....	13
1.3.4 在 Mac OS 中安装 Docker .....	15
1.3.5 在其他系统中安装 Docker .....	16
1.4 Docker 的优势 .....	17
1.4.1 革命性的虚拟化方案 .....	17
1.4.2 高效的容器技术 .....	18
1.4.3 社区的力量 .....	19
1.5 Docker 的应用场景 .....	20
1.5.1 超短时间部署运行 .....	20

1.5.2 节约迁移时间 .....	21
1.6 本章小结 .....	21
<b>第 2 章 镜像与仓库 .....</b>	<b>22</b>
2.1 镜像的概念 .....	22
2.1.1 联合文件系统 .....	22
2.1.2 Docker 中的镜像 .....	23
2.1.3 镜像的分层结构 .....	24
2.1.4 镜像的写时复制 .....	25
2.2 使用和管理镜像 .....	26
2.2.1 获取镜像 .....	26
2.2.2 列出镜像 .....	28
2.2.3 获得镜像的详细信息 .....	28
2.2.4 删除镜像 .....	31
2.2.5 镜像的迁移 .....	32
2.3 Docker Hub .....	33
2.3.1 镜像仓库 .....	33
2.3.2 Docker Hub 简介 .....	34
2.3.3 注册 Docker Hub 账号 .....	35
2.3.4 搜索镜像 .....	36
2.3.5 共享自动构建镜像 .....	38
2.4 搭建私有仓库 .....	40
2.4.1 镜像分发服务 .....	40
2.4.2 Docker Registry HTTP API .....	41
2.4.3 部署私有仓库 .....	42
2.5 本章小结 .....	44



第3章 管理和使用容器 .....	45
3.1 管理容器 .....	45
3.1.1 创建容器 .....	45
3.1.2 容器的启动过程 .....	48
3.1.3 列出容器 .....	49
3.1.4 容器的命名 .....	51
3.1.5 启动和停止 .....	52
3.1.6 暂停和恢复 .....	53
3.1.7 重启容器 .....	54
3.1.8 删除容器 .....	55
3.2 连接到容器 .....	55
3.2.1 查看进程信息 .....	56
3.2.2 查看容器信息 .....	56
3.2.3 容器日志 .....	62
3.2.4 衔接到容器 .....	63
3.2.5 在容器中执行命令 .....	64
3.3 容器的保存与迁移 .....	65
3.3.1 提交容器更改 .....	65
3.3.2 容器的导入/导出 .....	68
3.4 本章小结 .....	70
第4章 数据卷与网络 .....	71
4.1 数据卷 .....	71
4.1.1 关于数据卷 .....	71
4.1.2 数据卷的特点 .....	72
4.1.3 创建数据卷 .....	73
4.1.4 挂载数据卷 .....	74

4.1.5	删除数据卷 .....	76
4.2	数据卷容器 .....	77
4.2.1	关于数据卷容器 .....	77
4.2.2	创建数据卷容器 .....	78
4.2.3	连接数据卷容器 .....	79
4.2.4	数据卷的迁移 .....	80
4.3	网络基础 .....	82
4.3.1	网络简介 .....	82
4.3.2	查看网络配置 .....	83
4.4	网络访问 .....	85
4.4.1	宿主机端口映射 .....	85
4.4.2	容器连接 .....	87
4.5	本章小结 .....	90
第 5 章	制作镜像 .....	91
5.1	了解 Dockerfile .....	91
5.1.1	Dockerfile 简介 .....	92
5.1.2	使用 Dockerfile 创建镜像 .....	94
5.2	基础指令 .....	96
5.2.1	FROM .....	97
5.2.2	MAINTAINER .....	97
5.3	控制指令 .....	97
5.3.1	RUN .....	97
5.3.2	WORKDIR .....	99
5.3.3	ONBUILD .....	99
5.4	引入指令 .....	100
5.4.1	ADD .....	100

5.4.2 COPY .....	102
5.5 执行指令 .....	102
5.5.1 CMD .....	102
5.5.2 ENTRYPOINT .....	104
5.6 配置指令 .....	107
5.6.1 EXPOSE .....	108
5.6.2 ENV .....	108
5.6.3 LABEL .....	109
5.6.4 USER .....	110
5.6.5 ARG .....	111
5.6.6 STOPSIGNAL .....	112
5.6.7 SHELL .....	113
5.7 特殊用法 .....	113
5.7.1 环境变量 .....	113
5.7.2 指令解析 .....	114
5.7.3 忽略文件 .....	116
5.8 本章小结 .....	117

## 第二部分 实践篇

第6章 SSH 服务 .....	118
6.1 在 Docker 中使用 SSH .....	118
6.1.1 SSH 简介 .....	119
6.1.2 SSH 使用方法简介 .....	119
6.1.3 数据卷管理容器 .....	121
6.1.4 使用 SSH 服务容器 .....	122
6.2 构建 SSH 服务镜像 .....	124
6.2.1 构建方式比较 .....	124

6.2.2	通过提交构建	125
6.2.3	使用 Dockerfile 构建	127
6.3	本章小结	131
<b>第 7 章 Web 服务器</b>		<b>132</b>
7.1	Web 服务简介	132
7.1.1	万维网与网站	132
7.1.2	Web 服务	133
7.1.3	Web 服务程序	135
7.2	Apache	135
7.2.1	Apache 简介	135
7.2.2	安装 Apache	136
7.2.3	构建 Apache 镜像	139
7.2.4	测试 Apache 容器	142
7.3	Nginx	143
7.3.1	关于 Nginx	143
7.3.2	安装 Nginx	144
7.3.3	构建 Nginx 镜像	146
7.3.4	测试 Nginx 镜像	148
7.4	Tomcat	148
7.4.1	Tomcat 简介	149
7.4.2	安装 Tomcat	149
7.4.3	构建 Tomcat 镜像	152
7.5	本章小结	153
<b>第 8 章 数据库程序</b>		<b>155</b>
8.1	MySQL	155
8.1.1	MySQL 简介	156



8.1.2	安装 MySQL .....	156
8.1.3	构建 MySQL 镜像 .....	162
8.1.4	测试 MySQL 容器 .....	164
8.2	MongoDB .....	166
8.2.1	MongoDB 简介 .....	166
8.2.2	安装 MongoDB .....	167
8.2.3	构建 MongoDB 镜像 .....	171
8.2.4	测试 MongoDB 容器 .....	173
8.3	本章小结 .....	176
第 9 章	缓存工具 .....	177
9.1	Memcached .....	178
9.1.1	Memcached 简介 .....	178
9.1.2	安装 Memcached .....	179
9.1.3	构建 Memcached 镜像 .....	184
9.1.4	测试 Memcached 容器 .....	186
9.2	Redis .....	188
9.2.1	Redis 简介 .....	188
9.2.2	安装 Redis .....	188
9.2.3	构建 Redis 镜像 .....	193
9.2.4	测试 Redis 容器 .....	195
9.3	本章小结 .....	196
第 10 章	动态处理程序 .....	197
10.1	Java .....	197
10.1.1	Java 简介 .....	198
10.1.2	安装 Java .....	198
10.1.3	构建 Java 镜像 .....	204

10.1.4	测试 Java 容器	206
10.2	PHP	207
10.2.1	PHP 简介	207
10.2.2	安装 PHP	208
10.2.3	构建 PHP 镜像	214
10.2.4	测试 PHP 容器	216
10.3	Python	217
10.3.1	Python 简介	217
10.3.2	安装 Python	218
10.3.3	构建 Python 镜像	223
10.3.4	测试 Python 容器	224
10.4	Node.js	225
10.4.1	Node.js 简介	225
10.4.2	安装 Node.js	226
10.4.3	构建 Node.js 镜像	228
10.4.4	测试 Node.js 容器	230
10.5	本章小结	231
第 11 章	综合演练	232
11.1	演练目标	232
11.1.1	目标概述	232
11.1.2	代码编写	233
11.2	环境搭建	237
11.2.1	准备镜像	237
11.2.2	程序配置	239
11.3	项目运行	248
11.3.1	启动容器	248

11.3.2 测试项目 .....	249
11.4 本章小结 .....	253

### 第三部分 提高篇

第 12 章 网络进阶 .....	254
12.1 网络实现 .....	254
12.1.1 容器网络基础 .....	255
12.1.2 网络模型 .....	257
12.2 Docker 中的网络 .....	258
12.2.1 默认网络 .....	258
12.2.2 自定义网络 .....	261
12.2.3 容器与外部通信 .....	262
12.2.4 容器间通信 .....	264
12.3 网络实践 .....	265
12.3.1 管理容器网络 .....	265
12.3.2 容器连接网络 .....	267
12.3.3 配置 docker0 网桥 .....	269
12.3.4 自定义网桥 .....	271
12.3.5 配置 DNS .....	271
12.3.6 使用 IPv6 .....	273
12.4 本章小结 .....	274
第 13 章 安全加固 .....	275
13.1 深入理解 Docker 安全 .....	275
13.1.1 命名空间隔离 .....	276
13.1.2 资源控制组 .....	277
13.1.3 内核能力机制 .....	277

13.2	资源使用限制 .....	278
13.2.1	通过控制组限制 .....	278
13.2.2	通过 ulimit 限制 .....	280
13.2.3	网络访问限制 .....	280
13.3	校验与监控 .....	281
13.3.1	镜像签名 .....	281
13.3.2	运行状态监控 .....	283
13.4	联级防护 .....	284
13.4.1	组合虚拟化 .....	284
13.4.2	文件系统安全 .....	284
13.5	内核安全技术 .....	285
13.5.1	Capability .....	286
13.5.2	SELinux .....	287
13.5.3	AppArmor .....	288
13.6	本章小结 .....	289
第 14 章	Docker API .....	290
14.1	关于 Docker API .....	290
14.1.1	通用操作接口 .....	290
14.1.2	关于 RESTful .....	291
14.1.3	Docker API 的优势 .....	292
14.1.4	Docker API 的分类 .....	293
14.2	使用 Docker Remote API .....	293
14.2.1	关于 Docker Remote API .....	294
14.2.2	Docker Remote API 的版本 .....	299
14.2.3	通过 Remote API 列出容器 .....	300
14.2.4	通过 Remote API 列出镜像 .....	302



14.3 使用 Docker Registry API.....	303
14.3.1 关于 Docker Registry API.....	304
14.3.2 Docker Registry API 的主要功能.....	304
14.3.3 Docker Registry API 的版本.....	305
14.3.4 通过 Registry API 拉取镜像.....	306
14.3.5 通过 Registry API 推送镜像.....	307
14.4 本章小结.....	309
第 15 章 管理工具.....	310
15.1 Docker Compose.....	310
15.1.1 Docker Compose 简介.....	311
15.1.2 安装 Docker Compose.....	313
15.1.3 Docker Compose 配置文件.....	314
15.1.4 常用的 Docker Compose 命令.....	315
15.2 Docker Machine.....	318
15.2.1 Docker Machine 简介.....	318
15.2.2 安装 Docker Machine.....	320
15.2.3 Docker Machine 常见命令.....	321
15.3 Docker Swarm.....	322
15.3.1 Docker Swarm 简介.....	322
15.3.2 Docker Swarm 结构.....	323
15.3.3 使用 Docker Swarm.....	323
15.3.4 Docker Swarm 常见命令.....	325
15.4 本章小结.....	327
第 16 章 Docker 的技术架构.....	328
16.1 命名空间.....	328
16.1.1 关于 Linux 命名空间.....	328

16.1.2	命名空间的系统调用 .....	329
16.1.3	命名空间的分类 .....	330
16.2	控制组 .....	332
16.2.1	关于 Linux 控制组 .....	332
16.2.2	CGroups 的组成 .....	333
16.2.3	容器与控制组 .....	334
16.3	联合文件系统 .....	336
16.3.1	关于 UFS .....	336
16.3.2	Docker 中的 UFS .....	337
16.4	Docker Engine 架构 .....	338
16.4.1	Docker Engine 的组成结构 .....	338
16.4.2	Docker Daemon .....	339
16.4.3	Docker CLI .....	342
16.5	本章小结 .....	344

# 第一部分 基础篇

## 第 1 章

### 初识 Docker

在互联网浪潮中，无时无刻不涌现令人惊叹的新想法、新技术，其中不乏原理和实现简单却又能准确切中行业痛点、解决最实际的问题的优秀之作。这些具有时代性、跨越性的技术，总能引领和推动相关周边技术领域的快速发展，也会引发人们的热议和追捧。它们作为技术领域的弄潮儿，凝结了很多技术结晶。它们犹如数学家精心计算得出的公式，宛若艺术家手中婉转流畅的线条，极其简单却又甚为优雅，在并不复杂的实现里蕴藏着巨大的能量。

Docker 作为走在时代前列的技术之一，仅仅诞生数年就凭借其在虚拟化领域的整合与革新，蜚声云计算乃至整个科技界。那么 Docker 究竟是怎样的技术？它与虚拟化又有着怎样千丝万缕的联系？它能为我们的工作带来怎样的便利？在本章中，我们将展示 Docker 的前世今生，介绍 Docker 的组成结构，以及如何安装并在开发与运维中利用 Docker 提高工作效率。

## 1.1 虚拟化

在介绍 Docker 之前，我们先谈谈虚拟化，它是 Docker 的理论基础，也是 Docker 所实现功能的目的。

### 1.1.1 虚拟化技术

自计算机诞生起，程序开发者们就一直迫切地想解决一个难题——如何将编写的程序无缝地迁移到其他机器中正常地执行。程序的运行是通过 CPU 解析程序中的指令，并做出相应的操作来完成的，而不同厂商生产的不同型号的 CPU，使用的指令集往往有所区别，甚至完全不同。即使我们针对不同的 CPU 编译不同指令集的应用程序，在实际的运行中，不同的机器也会连接不同的硬件，运行不同的操作系统，有不同的共享链接库。更何况两台机器中相同的共享类库有着不同的版本时，非常有可能造成程序运行的不正常，这就使程序难以直接移植到其他机器中运行。

为了解决程序在迁移过程中出现的水土不服的问题，早在 20 世纪 60 年代，虚拟化的概念就被提了出来。虚拟化最早诞生于 IBM 对其实验性虚拟机系统的描述，在这个实验性系统中，IBM 的技术人员让操作系统根据给定的硬件环境，在主机中创造出一个供应用程序使用的虚拟计算机环境。虚拟的环境对应用程序隐藏了主机真实的硬件环境，取而代之的是一套完全相同的假环境。当应用程序在这个虚拟环境中运行时，所见和所使用的就是完全统一的硬件体系和软件依赖库，这就解决了程序在不同机器上运行的一致性问题。

虽然 IBM 很早就实现了虚拟化，但当时所设计的虚拟化方案是针对 IBM 的大型机业务的，在虚拟化的过程中需要专门的硬件设备，对硬件的依赖较高。这种只能针对特定硬件使用的虚拟化，让使用虚拟化的成本远高于对程序进行适配的成本，这使虚拟化技术在出现的早期并没有得到大范围的推广。

随着云计算时代的到来，越来越多的数据通过互联网传递到了云端，这就意味着在提供云服务的系统里，每时每刻都要处理几十亿、上百亿乃至更多的数据。而处理这些数据的程序，需要强大的资源，特别是硬件资源来驱动，如图 1-1 所示。



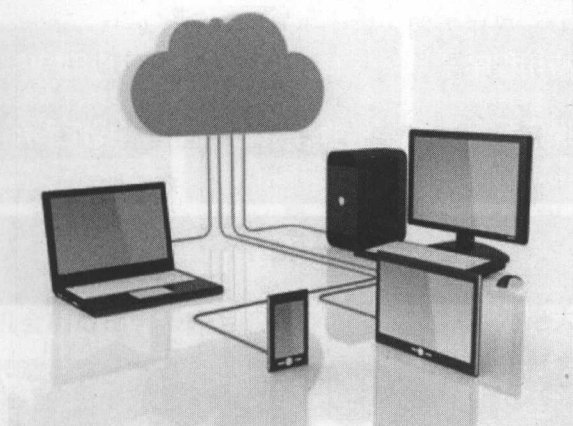


图 1-1 云计算

计算机技术的不断发展，特别是互联网和云计算时代的到来，逐渐推动了大型机向小型机的转移。过去，大型的网站或服务器都是以大型物理机运行的。随着越来越多的数据被传到云端，提供云服务的系统需要处理更多的数据。现在，制造硬件的速度已经远远追赶不上互联网和云计算发展的脚步，当业务发展需要服务提供更高的性能支持时，需要对大型机进行硬件升级，对于单一机器的硬件提升，其成本是指数倍增长的，这就造就了服务能力的瓶颈。好在互联网的发展，使大型服务可以通过分布在网络各端的小型机器协作来完成。通过大型机向小型机的转移，让提升性能的成本从指数增长变为了线性增长，有效降低了企业和开发者搭建大型服务的负担。这种将应用程序分散到不同的机器中，通过很多机器所组成的集群来协同完成的工作，即分布式计算。

随着越来越多的企业，特别是像 Google、Facebook 这种大型公司开始大面积采用小型机集群的方式搭建它们的服务，我们一开始提到的问题又被重新提到了台面上。众多的机器组成的集群，必然带来在每台机器中部署程序时的兼容问题，而能够让程序无缝运行在不同机器上的虚拟化技术，无疑是解决这个问题最好的方法。

我们可以把虚拟化技术作为计算机资源的一种管理技术，它掌控计算机中的 CPU、存储、网络等各种实体资源，并将这些资源抽象化，通过统一的形式展现给应用程序。也就是说，运行在虚拟环境中的程序，对接和使用的是虚拟化程序提供的资源，它们并没有接触真实的环境。在迁移到其他机器的时候，只要通过虚拟化程序制造出完全一样的虚拟环境，就能让应用程序的运行与之前完全一致。图 1-2 展示了通过虚拟化技术实现的分布式存储。

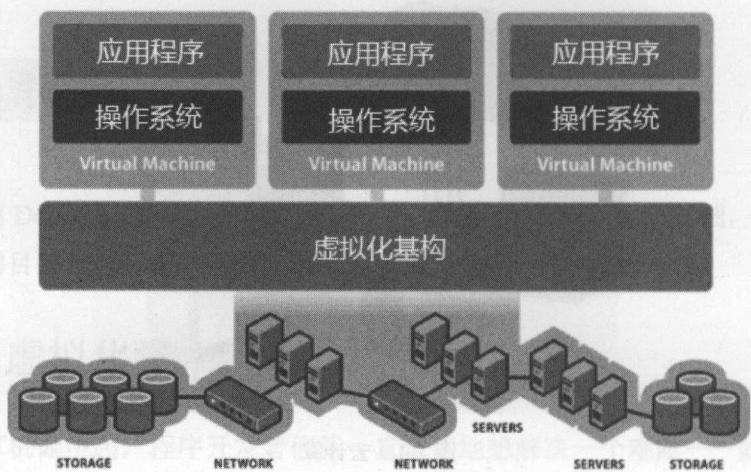


图 1-2 分布式存储的虚拟化实现

虚拟化技术将应用与真实的计算机资源分离，不但打破了应用程序与真实资源之间不可切割的障碍，也让资源跨物理或地域的配置和使用成为可能。通过虚拟化技术，可以更容易地完成应用程序对依赖资源的解耦，让应用程序轻松地运行在更多的环境之中。

### 1.1.2 虚拟化的分类

虚拟化是将物理主机的真实软硬件资源进行整合，形成一套统一的展示形式并制成虚拟环境。运行在虚拟环境中的程序，即使在不同的环境中，也因为调用了相同的接口，不会感觉到周围的运行环境发生了变化。对于我们来说，虚拟化只是定义了为程序隐藏真实环境的方式，真正实现虚拟化还需要具体的执行虚拟化过程的程序。

实现虚拟化的硬件和程序很多，但可以根据功能及实现方式，对它们进行归类。

IBM 最早实现的虚拟化方案，虚拟化过程是在 IBM 自己制造的硬件平台上完成的。这个硬件平台完成了对自身硬件资源的隐藏，取而代之的是程序需要的虚拟运行环境，这种方式通常被称为硬件虚拟化。由硬件完成的虚拟化，局限性比较大。因为实现虚拟化需要购置专门的硬件平台，并且这些硬件设备与其他普通硬件设备往往存在兼容问题，所以我们很难搭配其他硬件设备来使用。这就造成了这种用于虚拟化的硬件平台，很难进行硬件的升级与革新。当然，硬件虚拟化有它独特的好处。比如像 CPU 这样的硬件模块，若使用其他方式来实现虚拟化，往往会造成性能上的较大损失，而由硬件完成的虚拟化，对性能的影响就小很多。

除了硬件虚拟化，还有一种对虚拟化的软件实现，即虚拟机（Virtual Machine）。虚拟机通常在应用程序和硬件资源间搭建一个 Hypervisor 层。应用程序在虚拟化环境中对资源的调用，都是对 Hypervisor 的调用。因为 Hypervisor 提供了统一的调用接口，所以程序感觉不到周围的计算机资源发生了变化。而 Hypervisor 会将对它的调用，转换为真实物理主机上对应的资源调用方式。在虚拟机中，我们最熟悉的当属 Java 虚拟机（JVM，Java Virtual Machine）了，Java 应用程序正是通过 JVM 实现了跨平台运行。相较于 Java 虚拟机，更多的虚拟机会制造出一套虚拟的操作系统环境，因为绝大多数应用程序都是以操作系统作为开发和适配对象的，虚拟操作系统的做法就能让更多的程序在虚拟机中运行了。

在虚拟机的 Hypervisor 层之下，又有两种对接真实计算机资源的方法：一种是通过物理主机的操作系统间接访问计算机资源；另一种则是直接对接硬件资源。第二种方式其实和硬件虚拟化类似，但硬件虚拟化要求设备支持虚拟化，所以很难完全实现全部的硬件虚拟化。由于硬件虚拟化和软件虚拟化各有优势和劣势，在实际的生产中，通常会将这两种方式结合在一起使用。对于软件虚拟化方案会造成性能损失比较大的，而硬件虚拟化能够支持和弥补的部分，则交给硬件来完成虚拟化。这种软硬件配合完成虚拟化的方式，通常称为硬件辅助虚拟化。在 VMware Workstation、Xen、KVM 等大多数虚拟机程序中，在纯软件实现虚拟化的基础上，都支持硬件辅助虚拟化。图 1-3 展示了虚拟机主要的实现方式和各层次的关系。

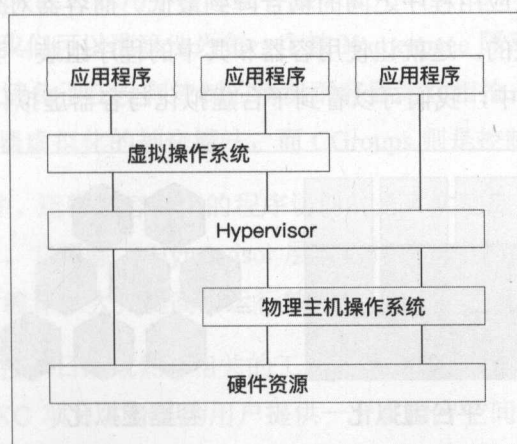


图 1-3 虚拟机的组织架构

除了上述的整体虚拟化方案，还有很多与虚拟化相关的概念，如网络虚拟化、服务虚拟化、数据库虚拟化等。这些领域都是利用虚拟化的概念，实现了对资源的汇总和再分配，细节这里就不在展开了。



## 1.2 容器技术与 Docker

容器技术是组成 Docker 的核心技术，其在 Docker 中扮演着举足轻重的角色。那么容器技术是怎样的技术，它与 Docker 又有怎样千丝万缕的联系？在本节中我们将进行具体介绍。

### 1.2.1 容器技术

在前一节我们谈到了多种虚拟化的分类，其中最具代表性的是通过 Hypervisor 实现的虚拟机，不过虽然虚拟机在跨平台部署上已经取得了不小的成效，但庞大的组织结构和低下的效率一直阻碍着其发展，因此，出现了虚拟化中另一大分支——容器技术。容器虚拟化技术是虚拟化的重要组成部分，也是本书的主角 Docker 赖以生存的基础，我们必须着重介绍。

容器技术就是将应用程序打包到每个单独的容器之中，通过这个封装的过程，容器技术将每个程序隔离开，打断了程序之间的依赖和连接关系。也就是说，一个庞大的服务系统在容器技术的支持下，可以由许多不同的应用程序所寄居的容器组合而成。这种拆解再构合的过程，让应用程序之间的耦合降到最低。而容器对外的接口就像集装箱之间的接槽一样，是一致的，这就让使用容器和其中的程序组装一套系统的过程变得极其简单和方便。在图 1-4 中，我们可以看到平台虚拟化与容器虚拟化的主要区别。

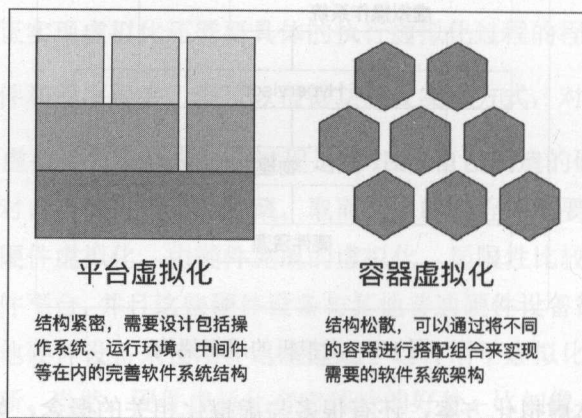


图 1-4 平台虚拟化与容器虚拟化的比较

容器技术不但实现应用程序对资源调用的封闭，还实现了应用程序相互之间调用的

解耦。通过容器技术，在开发和部署过程中，我们可以像搭积木一样轻松组合所需要的组件程序，这让虚拟化技术带来的便利又提升到了新的高度。

在容器虚拟化技术领域，家喻户晓的当属 LXC (Linux Containers) 了，它是由 Linux 内核提供支持的一项操作系统层级的虚拟化技术。当然，LXC 也是一项容器技术，它将不同的软件本身的代码，以及所需的操作系统核心库包裹在独立的容器里，并运行在各自的沙盒环境之中。比较特别的是，Linux 内核提供了一套专门用于支持容器技术的用户空间接口，其中最主要的是 Namespace 和 CGroups 这两项技术。

Namespace 即命名空间，为了防止与其他计算机领域的命名空间相混淆，通常加上前缀称之为“内核命名空间”。Linux Container 中的命名空间的性质和 C++命名空间相似，只不过在 Linux 里，用 Namespace 区分的是网络、PID、IPC、挂载目录等程序运行信息。通过 Namespace 可以很容易地将运行在同一个真实系统下的程序进行隔离，使不同 Namespace 中的程序是相互不可见的，这就达到了虚拟化隔离的效果。而 Namespace 所建立起的应用栅栏，其实就是容器的外壳，因此，Namespace 可以算是 Linux 核心虚拟化技术最重要的一部分。

CGroups 是控制群组 (Control Groups) 的简写，最早来源于 Google 工程师开发的进程容器，在 Kernel 2.6.24 版本中被加入到 Linux 中。CGroups 的目标是对系统资源进行管理。在 Namespace 的支持下，我们可以很容易地将程序隔离成不同的容器，或者说是应用群组，而 CGroups 正可以用于对这些容器所使用的 CPU、内存、IO 等资源进行控制。通过 CGroups，我们可以准确地为每一个被 Namespace 隔离的容器配置物理主机中真实的计算机资源，避免了在没有控制的情况下容器间互相抢占资源的情况。可以说，Namespace 是实现容器虚拟化的神奇魔法，而 CGroups 则是控制这个魔法的魔法杖。

在 LXC 的实现中，运行在容器中的程序访问的是真实物理主机的系统，与通过虚拟机实现的虚拟化相比，它消除了 Hypervisor 层，运行在其中的应用程序既不需要指令级模拟，也不需要即时编译，大大提高了运行效率。

Linux Container 技术日渐成熟，相关的工具也逐渐增多，与 LXC 同名的容器管理工具就是其中之一。LXC 项目本身是为用户提供一个在用户空间管理 LXC 容器的工具，其虚拟化实现依靠的仍然是 Linux Container 技术。LXC 项目封装了对容器的操作，但其在资源控制方面依赖于 CGroups 制定的框架，而在隔离控制方面依赖的是 Namespace 的特性。LXC 使 Linux Container 技术变得更加清晰，让用户可以更方便地使用 Linux 容器技术实现虚拟化。

## 1.2.2 Docker 简介

2013 年年初, 一家 PaaS 服务提供商 dotCloud 将其内部开发的 Docker 项目进行了开源, 从此拉开了 Docker 时代的序幕。短短几年间, Docker 已经变得家喻户晓, 成为脍炙人口的优秀容器虚拟化项目, 也有越来越多的组织和企业开始使用 Docker 构建和部署它们的应用和软件系统。

Docker 最初建立于 LXC 项目的基础上, 旨在进一步优化对容器操作的使用体验。Docker 在 Linux Container 技术的基础上, 将为系统设计容器转变成为软件或微服务设计容器。可以说, Docker 更强化了软件之间的隔离, 它提供了多种方便管理和使用容器的模块和工具, 使用者无须关注虚拟化实现的底层及中间结构, 只需对容器进行操作。

随着 Docker 的发展, dotCloud 公司索性放弃了其他业务, 将公司改名为 Docker Inc (图 1-5 为 Docker 的 Logo), 专门从事 Docker 及与其生态圈相关的工作。目前, Docker 已经加入 Linux 基金会, 代码也托管于 GitHub, 遵循 Apache License 2.0 开源协议。其他企业与个人, 也为 Docker 生态进行了软件等技术的贡献。这一切都让 Docker 逐渐成为轻量级虚拟化的代名词, 而 Docker 的发展速度, 也需要我们不断重新对它进行定义。



图 1-5 Docker

图 1-6 总结了虚拟机与 Docker 容器技术在集成结构上的区别。Docker 不但对上层软件的搭建和部署不断进行优化, 其影响也延伸到了底层结构的设计中。在较新的 Linux Kernel 中, 与 Linux 虚拟化相关的设计和实现, 已经受到了 Docker 的推动。逐渐成熟的 Docker 也摆脱了 LXC 项目的底层架构, 使用自己开发的 Libcontainer 作为底层容器技术的实现。

Docker 是一个典型的 C/S 结构的程序, 其主要功能都集成在 Docker Daemon 中。可以把 Docker Daemon 理解为 Docker 的 Server 端, 保证了容器技术的实现。在 Docker Daemon 的外层, 包裹着一层 REST API, 这些接口由 Docker Daemon 提供, 并可以使用它们对 Docker Daemon 进行操作。这些能够操作 Docker 服务的 API, 称为 Docker API, 在之后的章节中我们会专门进行介绍。Docker 软件不仅包含了 Docker Daemon 服务端, 还打包进了 Docker CLI 客户端程序。Docker CLI 将 Docker REST API 中的操作中继到命令行中的命令, 这就是使用 Docker 所输入的命令了。当我们启动 Docker 服务时, 其实



启动的是 Docker Daemon, 而当我们使用 Docker 命令进行操作时, 其实是控制 Docker CLI 通过 DockerREST API 发送控制命令到 Docker Daemon。

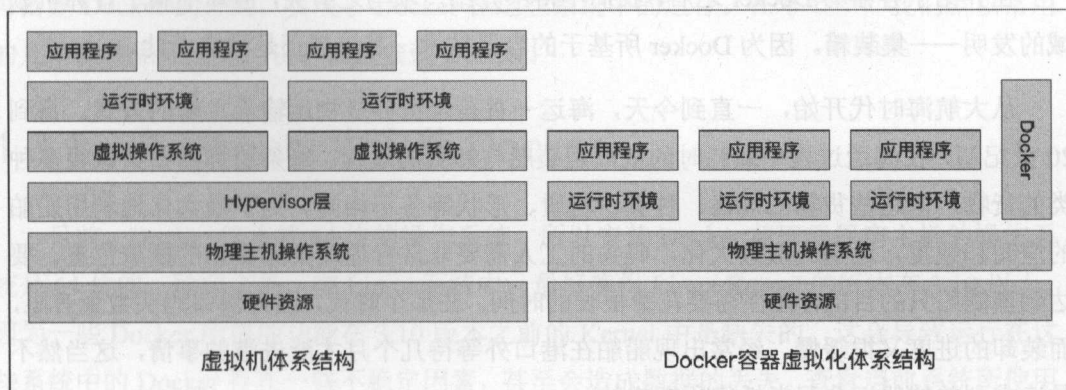


图 1-6 虚拟机与 Docker 容器技术的体系结构

如图 1-7 所示, 在 Docker 所提供的容器技术中, 主要有镜像、容器、数据卷、网络四大核心模块。镜像 是容器所打包的应用程序、系统类库等运行时环境的文件, Docker 为镜像提供了一种类似 Git 的版本管理方式, 这种形式实现了镜像之间相同的基础部分的共享, 也大幅减少了镜像所占用的存储空间。容器也是 Docker 中重要的概念, 在 Docker 中, 容器代表的是被容器技术所隔离的运行时环境。另外, Docker 还提供网络、数据卷等其他辅助模块, 为容器提供网络连接、数据存储等层面的支持。上述组件都可以通过 Docker CLI 的 docker 命令进行配置和操作, 这些模块的介绍和使用方法, 我们都将在之后的章节中逐一展开讲解。

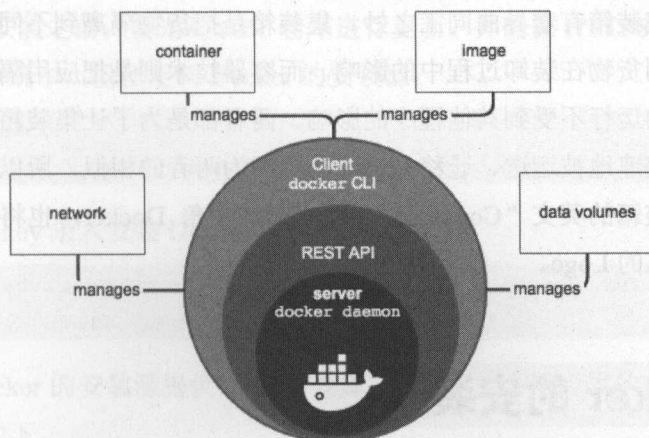


图 1-7 Docker 程序的组成

### 1.2.3 改变世界的发明

在介绍了容器与 Docker 之后，我们不得不介绍一项与之有关，但却不属于计算机领域的发明——集装箱。因为 Docker 所基于的容器技术，其思想正是源自于此。

从大航海时代开始，一直到今天，海运一直是全世界货物运输最主要的方式。直到 20 世纪初，在海运过程中最耗时的依旧是装船与卸货的过程。每条船都需要搭载很多种类的货物，但这些货物的规格、种类、材质、形状等各不相同。为了最大化地利用船舶的空间和载重，达到利益最大化，码头的工人需要非常合理地安排货物的摆放位置。要达到塞满船只的目的，往往需要花费很长的时间。在那个时代，因为港口码头数量有限，而装卸的进度又很缓慢，经常出现船舶在港口外等待几个月才能进港的事情，这当然不是追求利益的船运公司老板们所期望的。

马尔科姆·麦克莱恩就是希望改进缓慢的船舶装卸过程的船运公司老板之一。通过对其他运输行业的总结研究，他设计了一套规格完全一致大的箱子。因为这些箱子的大小完全相同，所以在装卸的过程中就省去了设计安排摆放位置的工作。有了这套箱子，装卸时间从原来的几个月渐渐减少到了几天，甚至几个小时。有人曾这样描述这套箱子带来的变化：以前的港口码头周围都是酒吧和风月场所，因为水手们在货物装卸的过程中实在百无聊赖。而有了这套箱子之后，几乎水手们都还没有下船，货物就已经装好了，只好再次启程。这套箱子就是集装箱。随着集装箱被广泛使用，全球的货物流动也随着船运业效率的提高变得更加迅速。这推动了全世界的交流与发展，迈出了全球化的第一步。这也是集装箱这个没有技术含量的发明，被推崇为 20 世纪最伟大的发明之一的原因。

容器技术与集装箱有着异曲同工之妙。集装箱是把货物隔离到不同的箱子中，通过这种方式消除不同货物在装卸过程中的影响。而容器技术则是把应用隔离到不同的容器中，让应用程序的运行不受到其他程序的影响。两者都是为了让集装箱中的货物或容器中的程序，能够快速地被运送、迁移。也许正是因为两者的相似，所以“容器”这个词的英文就取自集装箱的英文“Container”。而本书的主角 Docker，也将大鲸鱼背集装箱这个图案用作自己的 Logo。

## 1.3 Docker 的安装

要使用 Docker，就必须先安装 Docker。目前，Docker 已经支持在绝大多数 Linux



系统中安装或者编译安装并使用，只要它们使用的 Linux Kernel 版本能够提供 Docker 所需要的基础能力即可。另外，在其他主流的操作系统中，Docker 也通过虚拟机运行 Linux 再搭建 Docker 的方式，实现了 Docker 在这些系统中的运行。在本节中，我们就以常用的几个操作系统为例，讲解如何安装 Docker。

### 1.3.1 在 Ubuntu 中安装 Docker

目前，Docker 仅支持 64 位的操作系统，所以安装 Docker 前请确保宿主机的操作系统是 64 位的。除此之外，在 Linux 系统中，最好确保 Linux Kernel 的版本在 3.10 以上，因为一些 Docker 所需的功能在 3.10 版本之前的 Kernel 中是缺失的，这会导致运行在这些系统中的 Docker 存在一些不稳定因素，甚至会造成数据的丢失。查看当前系统所使用的 Kernel 版本的方法是：

```
$ uname -r
```

在结果中我们可以看到 Kernel 的版本，如果版本较低，建议先升级再进行 Docker 的安装。

Docker 官方为四个版本的 Ubuntu 系统提供了通过 APT 安装的支持，分别是 Ubuntu Xenial 16.04 (LTS)、Ubuntu Wily 15.10、Ubuntu Trusty 14.04 (LTS)、Ubuntu Precise 12.04 (LTS)。官方的支持基本覆盖了目前最常见的 Ubuntu 系统，所以我们建议通过 apt-get 来安装 Docker。另外，Docker 也存在于 Ubuntu Utopic 14.10 和 15.04 版本的 APT 应用仓库中，不过它们并不由官方提供支持。

在进行 APT 安装之前，要先对 APT 仓库进行更新，并确保 HTTPS 和 CA 证书模块被安装到当前的系统中，以便定制 Docker 的安装源：

```
$ sudo apt-get update
$ sudo apt-get install apt-transport-https ca-certificates
```

接着通过 apt-key 录入安装 Docker 所需的 GPG key：

```
$ sudo apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80 --recv-keys
58118E89F3A912897C070ADB76221572C52609D
```

之后要将 Docker 的安装源提供给 APT 仓库，Docker 为不同版本的 Ubuntu 系统所提供的安装源地址如下。

- ❑ Ubuntu Precise 12.04 (LTS): deb [https://apt.dockerproject.org/repo ubuntu-precise main](https://apt.dockerproject.org/repo/ubuntu-precise-main)。

- ❑ Ubuntu Trusty 14.04 (LTS): deb [https://apt.dockerproject.org/repo ubuntu-trusty main](https://apt.dockerproject.org/repo/ubuntu-trusty-main)。
- ❑ Ubuntu Wily 15.10: deb [https://apt.dockerproject.org/repo ubuntu-wily main](https://apt.dockerproject.org/repo/ubuntu-wily-main)。
- ❑ Ubuntu Xenial 16.04 (LTS): deb [https://apt.dockerproject.org/repo ubuntu-xenial main](https://apt.dockerproject.org/repo/ubuntu-xenial-main)。

选择符合当前系统的源地址,编写一个源地址文件并写入其中,这里我们使用 `docker` 作为安装源配置文件的名称:

```
/etc/apt/sources.list.d/docker.list
```

添加安装源后,Ubuntu 系统为安装 Docker 所做的准备工作就完成了。再次进行 APT 仓库的更新,卸载 Docker 的早期版本并校验安装源:

```
$ sudo apt-get update
$ sudo apt-get purge lxc-docker
$ apt-cache policy docker-engine
```

所有准备工作完成之后,就可以安装 Docker 了。Docker 在 APT 仓库中是以 `docker-engine` 这个名字出现的,安装时需要注意:

```
$ sudo apt-get install docker-engine
```

安装完成后,启动 Docker 服务才能让 Docker 运行起来:

```
$ sudo service docker start
```

## 1.3.2 在 CentOS 中安装 Docker

目前,Docker 官方只支持 CentOS 7.x 及以上版本,之前版本的系统安装的 Docker 版本会比较低。另外,在 Scientific Linux 等其他源于 RHEL7 版本的系统中,也有第三方提供和支持的 Docker。

在 CentOS 中,我们可以通过 Yum 很方便地安装 Docker。在安装之前,我们先要更新 Yum,以确保系统本身和其中所有的软件都是最新的:

```
$ sudo yum update
```

接着将 Docker 的安装源写入 Yum 的软件仓库配置中:

```
$ sudo tee /etc/yum.repos.d/docker.repo <<-'EOF'
[dockerrepo]
name=Docker Repository
baseurl=https://yum.dockerproject.org/repo/main/centos/7/
```

```
enabled=1
gpgcheck=1
gpgkey=https://yum.dockerproject.org/gpg
EOF
```

准备工作完成后,即可进行 Docker 的安装。Docker 在 Yum 仓库中的名称为 `docker-engine`,在较早版本的 CentOS 中也以 `docker-io` 的名字出现。

```
$ sudo yum install docker-engine
```

安装完成后,需要启动 Docker 服务才能让 Docker 运行起来:

```
$ sudo service docker start
```

小提示:在 Yum 仓库中,还有另外一款完全不相关的软件——`docker`,我们在安装 Docker 时要特别注意需要安装 Yum 仓库中的 `docker-engine`,不要安装成其他软件。

### 1.3.3 在 Windows 中安装 Docker

随着 Docker 的发展,Docker Inc 也为 Windows 平台提供了支持。在 Windows 平台上使用 Docker,能够享受到与其他平台一致的体验,也就是说,我们可以很方便地在 Windows 平台上进行与 Docker 相关的软件开发与测试,并且迁移到其他平台时,容器内部的环境不需要任何改动。

因为 Docker 依赖于 Linux Kernel 所提供的容器虚拟化技术的支持,所以在 Windows 平台上使用 Docker 需要先虚拟出一个 Linux 系统。在 Docker for Windows 较早的测试版本中,使用的是 VirtualBox 为 Docker 提供的虚拟 Linux 环境,随着研发的推进,目前已经切换到性能和契合度更佳的由微软提供的 Hyper-V 技术来实现虚拟的 Linux 环境。不过,安装和使用 Docker for Windows 时,无须关心这些细节,Docker 官方提供的安装包已经包括了所有的环境搭建。

到目前为止,Docker for Windows 仅支持 64 位 Windows 10 的专业版、企业版和教育版,并且需要安装较新的系统。Docker 官方声明,会在以后提供对更多版本的 Windows 10 系统的支持,但如果你仍然没有升级到 Windows 10,可以先进行系统的升级。如果你需要在 Windows 10 之前的系统中使用 Docker,可以到 Docker 的官方网站下载 Docker Toolbox。使用 Docker Toolbox 安装的 Docker 没有提供 Hyper-V 技术,依然使用虚拟机实现虚拟 Linux。

要进行 Docker for Windows 的安装,只需要在 Docker 的官网下载 Docker for Windows



的安装包并进行安装即可。因为有 Hyper-V 技术的加持, Docker for Windows 能更好地兼容和嵌入到 Windows 系统中, 也就是说, 运行在 Windows 平台的 Docker, 在性能上并不比在 Linux 里运行的 Docker 逊色太多。

在 Windows 中使用 Docker, 首先需要启动 Docker for Windows, 并等待 Docker 服务运行起来, 在这个过程中可以通过系统托盘中的 Docker 图标(见图 1-8)识别 Docker 服务的状态。当 Docker for Windows 正常运行后, 就能通过 PowerShell 控制台进行命令操作了。如图 1-9 所示, 我们可以通过命令查看 Docker for Windows 中包含的 Docker 的版本信息。

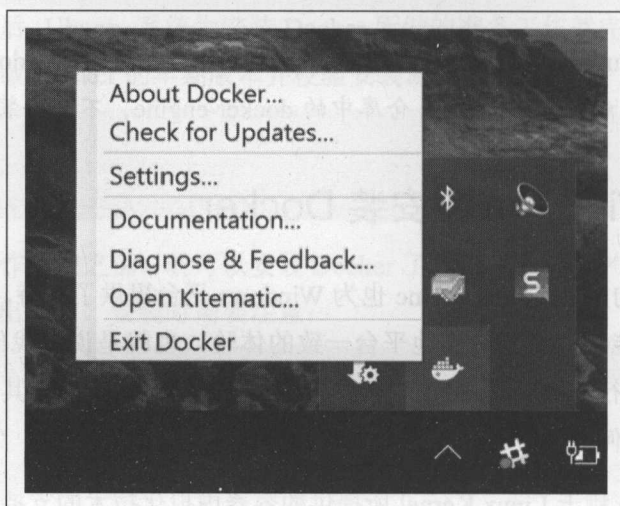


图 1-8 Docker 在 Windows 状态栏中的菜单

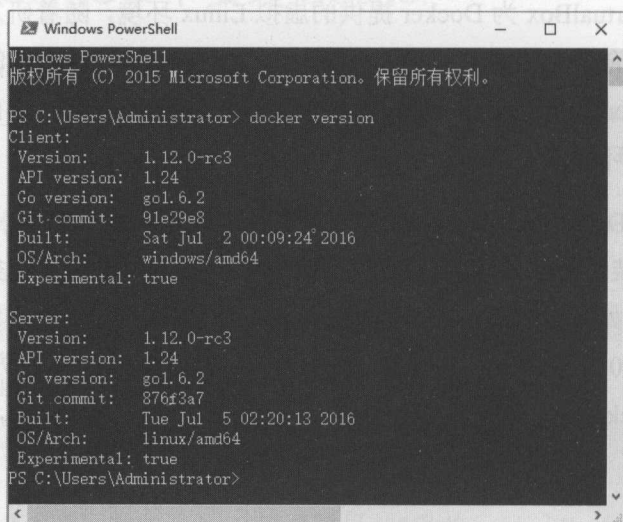


图 1-9 在 Windows 中查看 Docker 的版本信息



### 1.3.4 在 Mac OS 中安装 Docker

Docker 在 Mac OS 平台也得到了实现，在 Mac 平台安装 Docker，需要 Mac OS 的版本至少为 Yosemite，内存至少为 4GB。另外，Mac 也应该使用 2010 年或者更新的型号，因为这些型号的 Intel 硬件才支持 MMU（内存控制单元）和 EPT（扩展页表）。

最新的 Docker for Mac 基于 Mac OS 的 HyperKit 虚拟化技术，其与 Docker 在 Windows 中使用的 Hyper-V 技术相似，通过系统原生的虚拟化降低虚拟化 Linux 时 Hypervisor 层的复杂程度，从而优化了 Docker for Mac 运行时对系统资源的消耗。

从 Docker 官网下载 Docker for Mac 的压缩包 Docker.dmg 并打开，如图 1-10 所示，将 Docker 程序拖入应用程序文件夹，这样就能完成 Docker 的安装。

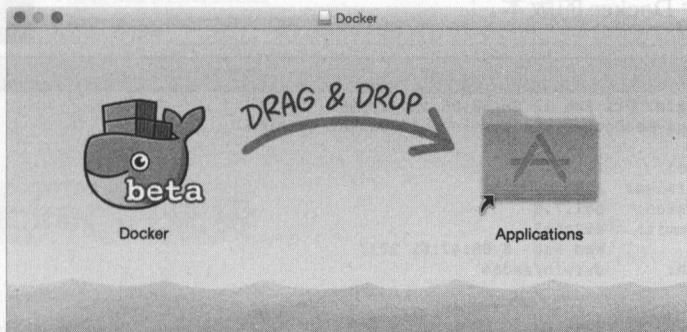


图 1-10 将程序拖入应用程序文件夹

打开 Docker for Mac 应用，Docker 就会运行起来。如图 1-11 所示，在 Mac OS 顶部的状态栏中，可以发现 Docker 的小鲸鱼图标，单击图标可以进行配置、反馈、检查更新等操作。

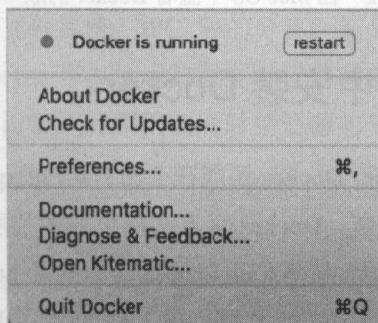


图 1-11 Docker 在 Mac 状态栏中的菜单

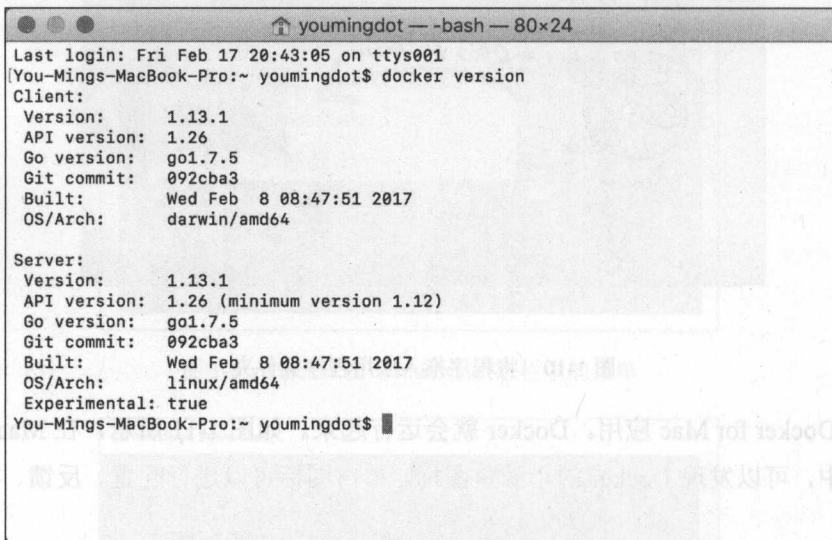
Docker for Mac 中除了提供 Docker Engine 和 Docker CLI, 还附带了 Docker Compose 和 Docker Machine, 我们可以分别通过以下命令查看各个组件的版本。

```
$ docker --version
Docker version 1.12.0-rc2, build 906eacd, experimental

$ docker-compose --version
docker-compose version 1.8.0-rc1, build 9bf6bc6

$ docker-machine --version
docker-machine version 0.8.0-rc1, build fffa6c9
```

当状态栏中显示的 Docker 状态转变为 Docker is running 时, 就能通过在 Mac 的终端 (Terminal) 中输入 docker 命令来使用 Docker 了。如图 1-12 所示, 我们可以通过 docker version 命令查看 Docker 的版本。



```
youmingdot ~ -bash - 80x24
Last login: Fri Feb 17 20:43:05 on ttys001
[You-Mings-MacBook-Pro:~ youmingdot]$ docker version
Client:
 Version:      1.13.1
 API version:  1.26
 Go version:   go1.7.5
 Git commit:   092cba3
 Built:        Wed Feb  8 08:47:51 2017
 OS/Arch:      darwin/amd64

Server:
 Version:      1.13.1
 API version:  1.26 (minimum version 1.12)
 Go version:   go1.7.5
 Git commit:   092cba3
 Built:        Wed Feb  8 08:47:51 2017
 OS/Arch:      linux/amd64
 Experimental: true
You-Mings-MacBook-Pro:~ youmingdot$
```

图 1-12 在 Mac OS 中查看 Docker 的版本信息

### 1.3.5 在其他系统中安装 Docker

除了上述提到的几个系统, Docker 还能在 Red Hat Enterprise Linux、Debian、Gentoo、Oracle Linux 等系统中进行安装。Docker 官方文档中列举了所有 Docker 官方支持的系统及在它们中安装 Docker 的流程, 读者们可以通过访问网址 <https://docs.docker.com/engine/installation/> 来获取 Docker 安装的详细教程。图 1-13 展示了 Docker 官方文档中关于在各个支持的系统中安装 Docker 的页面。



图 1-13 Docker 官方文档中关于安装的页面

## 1.4 Docker 的优势

Docker 从诞生到如今的炙手可热，不过几年的光景。能够让 Docker 迅速扩散并受到大家的赞许，很大一部分原因在于 Docker 相较于以往的虚拟化技术有着质的飞跃。

### 1.4.1 革命性的虚拟化方案

Docker 作为一款优秀的容器集成软件，和传统的容器虚拟化工具相比，有着非常多的改进与优化。采用 Libcontainer 作为容器技术支撑的 Docker，通过 Linux 内核命名空间实现了程序进程、网络、文件系统、IPC 等的分离，充分保障了容器的隔离性。而通过使用 Linux 控制群组，Docker 可以让用户根据自己的需求，将已经虚拟的 CPU、内存等硬件资源进行配置，在资源被充分利用的同时使资源的分配更可控。

重新定义的镜像技术，更是为 Docker 增光添彩。Docker 不改变基础镜像只进行上层写操作的增量镜像技术，大大增加了基础镜像可共享的内容。这不但可以让更多的容器共享同一个基础镜像，还减少了迁移过程中所需要传输镜像的体积，并且避免了修改容器和写入镜像时多次重复操作。



由于 Docker 容器对应用程序运行的沙盒环境做了良好的封装，以及镜像技术和所基于的联合文件系统的加持，Docker 容器拥有远超其他容器技术的迁移性，使用 Docker 可以真正实现在不同环境中的无差别部署。Docker 不但通过虚拟化将应用程序和硬件资源进行了切割，还让容器技术缩小到为应用服务的层面，也让运行在容器中的应用程序相互之间的依赖降至最低，这非常符合当下流行的微服务与分布式组合的技术架构。

在安全性上，有着内核级隔离的 Docker 容器，可以保障运行在容器之中的应用程序，不会干扰其他容器中的应用程序的运行，也不会受到其他容器中应用程序及其依赖库的影响，这样可以有效防范漏洞的产生并抑制漏洞的扩散。因为它高度解耦了应用程序间的关系，当遇到故障或者问题时，我们可以很容易地通过替换问题模块所在的模块来进行修复，其他模块的程序和相关依赖不会受到替换过程的影响，整个过程就像更换汽车零部件一样。

### 1.4.2 高效的容器技术

Docker 基于 Linux Container 技术，在运行过程中需要额外的 Hypervisor 层的支持，可以实现更高的性能和效率。在容器技术的加持下，Docker 容器的启动可以做到秒级的耗时，相较于传统的虚拟化技术，可以说是飞跃式的提升。因为是内核级的虚拟化，所以 Docker 在利用系统资源方面也有不俗的表现。表 1-1 为 Docker 容器与虚拟机在性能上的比较。

表 1-1 Docker容器与虚拟机的性能对比

特    征	容    器	虚  拟  机
硬盘使用	MB级	GB级
性能	接近原生	较低
启动速度	秒级	分钟级
单机支持量	上千个	几十个

由于 Docker 的设计是为独立应用封装容器，所以我们可以很轻松地通过 Docker 容器组装需要的服务系统体系，整个过程省去了大量应用程序搭建和调试的时间。使用 Docker 时，用很小的修改就能代替我们之前所做的大量工作，节约了大量时间。



### 1.4.3 社区的力量

Docker 最大的优势在于其拥有强大的社区力量，Docker 的快速发展离不开这些社区的推动。

在 Docker 逐渐受到关注的同时，越来越多的企业及组织开始选择使用 Docker 来搭建和部署它们的网站、服务器或者应用。而这些企业和组织中拥有强大生命力的开发者们，根据自己的需求提出和实现了对 Docker 的改进，并开发出很多能够很好地提高效率或解决问题的软件。这些企业、组织及开发者们，逐渐聚集形成了 Docker 的开发社区，讨论、交流并贡献出许多软件作品。这些改进软件以及优秀的周边软件，又被人们使用到生产中去，形成了 Docker 生态的闭环。图 1-14 展示了 Docker 生态的良性循环结构。

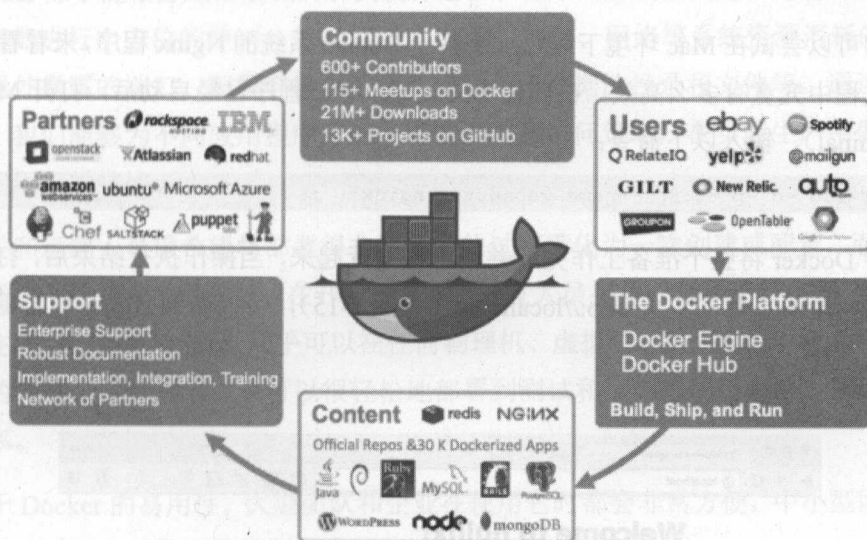


图 1-14 Docker 生态圈

众多 Docker 社区的存在，使得任何人使用 Docker 都非常简单，绝大多数问题也都能在 Docker 社区中找到答案。而许多用来辅助 Docker 的工具软件，也从社区中源源不断地涌现出来。在 Docker 中，你不需要担心有问题无法解决，不需要担心已有的程序含有 bug，亦不需要担心缺乏实用的工具，Docker 在众多社区支持下的发展速度，远高于我们学习 Docker 的速度。我们无法保证在不久的将来，Docker 生态的状况及使用方法是不是本书中所谈到的这样。

## 1.5 Docker 的应用场景

结合 Docker 的技术结构特点和优势，我们来谈谈 Docker 的使用场景，以及如何把 Docker 的优势应用到实际生产中，以解决已有的问题，并提高工作效率。

### 1.5.1 超短时间部署运行

Docker 最大的优势在于其充分发挥了容器技术的封装性，使用 Docker 可以轻松实现对应用程序及其依赖环境的迁移。

我们可以尝试在 Mac 环境下运行一个基于 Debian 系统的 Nginx 程序，来看看 Docker 在部署过程中究竟有多么高效。在确保 Docker for Mac 程序已经启动后，打开 Mac 的终端（Terminal），输入以下命令：

```
$ docker run -d -p 80:80 --name webserver nginx
```

等待 Docker 将整个准备工作完成并让容器运行起来，当操作执行结束后，打开浏览器，进入本地服务器的主页 <http://localhost/>（见图 1-15），可以看到 Nginx 服务器已经运行起来了。

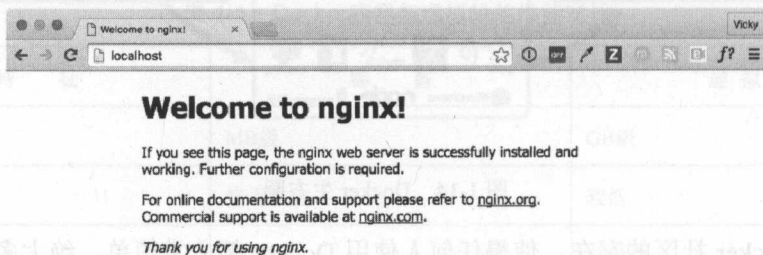


图 1-15 Nginx 服务器开始运行

仅通过一个命令就能完成软件从 Debian 平台部署到 Mac OS 平台的过程，不需要根据本机环境进行配置，由此可以看出 Docker 对提升部署效率的帮助是非常大的。

## 1.5.2 节约迁移时间

假设现在需要搭建最常见的 LAMP (Linux + Apache + MySQL + PHP) 的服务器架构。按照以往的做法, 需要分别安装 Apache、MySQL、PHP 以及各自运行的依赖库或软件。安装完成后, 还需要分别对它们进行配置, 调整它们之间的兼容性, 解决出现的问题。还需要进行程序与软件的联合调试, 确保程序和软件之间没有兼容问题。整个过程相当烦琐, 特别是开发环境与部署环境本身就具有一定的差异, 使兼容性问题特别容易出现。通常这些问题都不容易解决, 这让部署过程在时间上有了很大的不确定性, 如果整个应用体系庞大, 拥有非常多的应用以及微服务, 那么出现的问题就更难以处理了。

Docker 除了能很好地帮助我们解决软件迁移和程序间解耦的问题, 还能对应用程序所在的容器进行全方位的控制, 包括容器对 CPU、内存、网络等系统资源消耗的限制, 指定容器外暴露的端口, 配置容器内用于存储和共享数据的目录和文件等。通过这些控制手段, 我们能够为不同应用程序分配不同的资源和访问策略, 在实际生产中做到对每一个应用程序的掌控。

开发和运维人员最希望将部署得非常烦琐的过程简化为一次创建或配置, 而 Docker 就是完成这个愿望最佳的选择。在开发过程中, 开发人员搭建一套用于开发的容器集合, 在开发完成后, 利用 Docker 几乎可以在任何物理机、虚拟机、个人电脑、服务器等机器上运行的特性, 这套容器体系可以很轻松地部署到测试和运维的环境中去, 从而提高了工作效率。

由于 Docker 的易用性, 大型团队和企业在使用它时都会非常方便, 中小型团队使用它时可以感受到它所带来的便利。

## 1.6 本章小结

通过本章的描述, 相信读者们已经对 Docker 及其相关技术有了认识, 并了解了这些技术的发展史和现状。虚拟化的发展与演进, 得益于不断有引领其发展的技术出现, 当下, 这个作为领头羊的技术非 Docker 莫属。了解 Docker, 不仅对掌握虚拟化技术有很大的帮助, 在日常的开发、部署和运维过程中, 也会给我们带来极大的便利。在接下来的章节中, 我们就将细致地讲解 Docker 和其相关技术及工具的使用。



# 第 2 章

## 镜像与仓库

和大部分虚拟化机制一样，Docker 中也有镜像的概念，镜像是虚拟运行环境中硬盘数据的副本，是运行虚拟环境的基础，也是迁移虚拟环境有力的辅助工具。虽然都是镜像，但 Docker 中的镜像比以往的虚拟化技术中的镜像有很大的改进与提升。本章我们就来着重介绍 Docker 中的镜像，看看 Docker 镜像独特的结构，了解如何使用 Docker 进行镜像的创建、转移、删除等操作，并了解如何使用网络来分享我们制作的镜像。

### 2.1 镜像的概念

镜像是 Docker 文件系统的基础，也是 Docker 容器的根基，通过镜像来切入 Docker 的学习和使用无疑是最佳的方案。那么，Docker 的镜像与其他虚拟化体系中的镜像有何区别？Docker 的镜像系统又有哪些革命性的创新与创造呢？我们在这一节中就逐一展开，对 Docker 镜像进行诠释。

#### 2.1.1 联合文件系统

在庞大的开源社区驱动下，Linux 生态圈不断涌现出优秀的开源软件和设计思路，联合文件系统（UnionFS）就是其中之一。联合文件系统是一个轻量级、高性能的分层文件系统，它的特点就是支持将文件的修改变换为一层层的增量提交，并且支持将多个



不同的文件系统挂载到一个统一的虚拟文件系统下。联合文件系统能够将多个来自不同文件系统的文件或者目录组合到这个联合文件系统中，供应用程序访问，如图 2-1 所示。

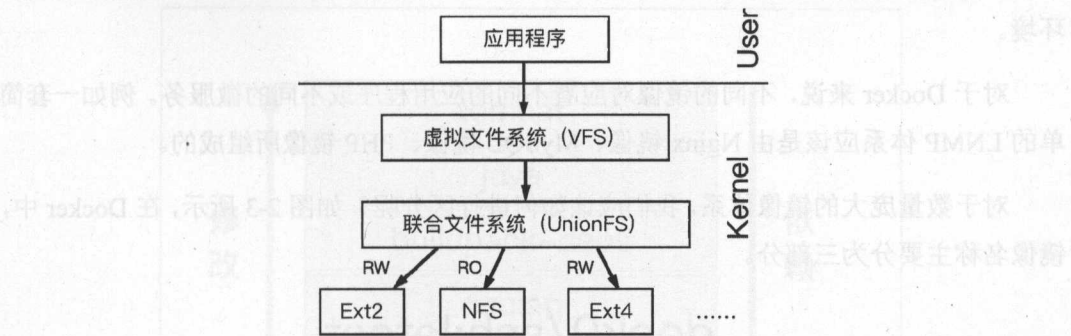


图 2-1 联合文件系统

Docker 利用联合文件系统能够组合挂载的特性，建立了一套文件系统分层体系。在这套体系中存储着每一个文件的修改历史。如图 2-2 所示，对文件所进行的更新、删除等其他修改操作，都不是直接作用于被修改的文件上，而是将修改后的文件直接通过联合文件系统的挂载机制替换掉实际访问的文件。而被修改的文件只是不能被程序和用户访问到，但仍然存在于存储器之中。

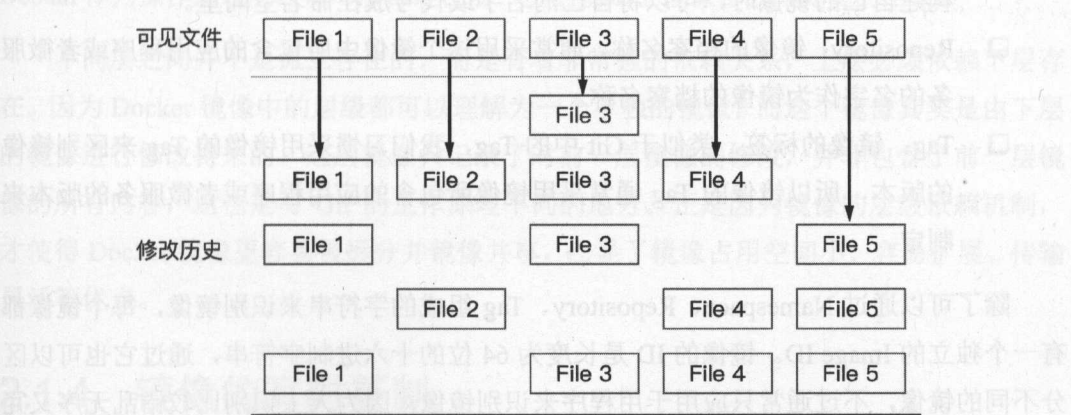


图 2-2 分层与文件修改历史

### 2.1.2 Docker 中的镜像

在 Docker 中，镜像是一个包含应用程序及相关依赖库的文件，在 Docker 容器启动

的过程中，它以只读的方式被用于创建容器运行的基础环境。如果把容器理解为应用程序运行的虚拟环境，那么镜像就可以被看作这个环境的持久化副本。通过镜像，我们可以很容易地保存虚拟环境的运行状态，并可以很方便地镜像迁移及反复构造相同的运行环境。

对于 Docker 来说，不同的镜像对应着不同的应用程序或不同的微服务。例如一套简单的 LNMP 体系应该是由 Nginx 镜像、MySQL 镜像、PHP 镜像所组成的。

对于数量庞大的镜像体系，我们应该如何进行区分呢？如图 2-3 所示，在 Docker 中，镜像名称主要分为三部分。

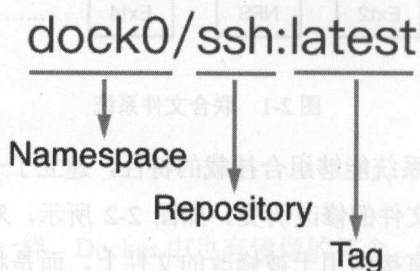


图 2-3 镜像的名称

- ❑ **Namespace:** 镜像的命名空间，用于区别构建镜像的组织或个人，所以我们在构建自己的镜像时，可以将自己的名字或代号放在命名空间里。
- ❑ **Repository:** 镜像的档案名称，通常采用这个镜像中所包含的应用程序或者微服务的名字作为镜像的档案名称。
- ❑ **Tag:** 镜像的标签，类似于 Git 中的 Tag，我们习惯采用镜像的 Tag 来区别镜像的版本，所以镜像的 Tag 通常采用镜像所包含的应用程序或者微服务的版本来制定。

除了可以通过 Namespace、Repository、Tag 组成的字符串来识别镜像，每个镜像都有一个独立的 Image ID。镜像的 ID 是长度为 64 位的十六进制字符串，通过它也可以区分不同的镜像，不过通常只应用于用程序来识别镜像，因为人工识别比较错乱无序又冗长的字符串，确实是一件费力不讨好的事情。

### 2.1.3 镜像的分层结构

与其他虚拟化体系中的镜像不同，Docker 的镜像是一个多层结构，镜像的每一层都是在原有层的基础上进行改动的。镜像的分层机制与 Git 的版本控制原理类似，每层镜

像都可以被视为一个提交，并且拥有独立的 ID，最顶层的 ID 就被视为镜像的 ID。

图 2-4 展示的是 tomcat:jre8 镜像的层级关系。

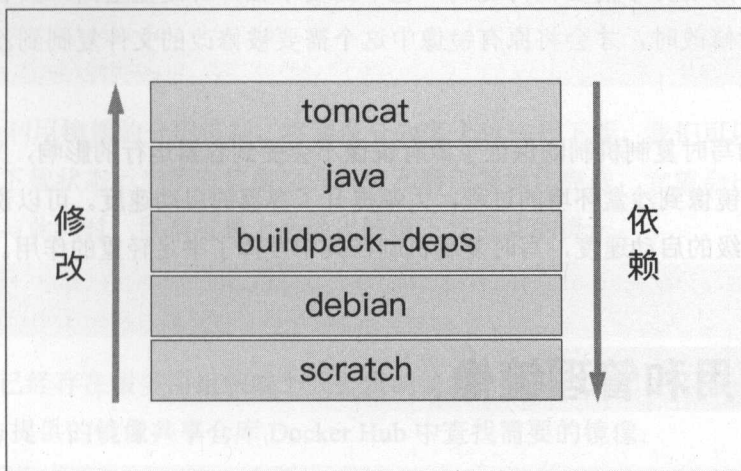


图 2-4 镜像层级依赖

这里的 scratch 是 Docker 提供的一个基础层，通常被称为 bootfs，即启动文件系统，通常情况下，我们不会直接和这一层打交道。而在 bootfs 之上，通常就是操作系统层，Docker 官方提供了很多常用的操作系统，如 Ubuntu、CentOS、Debian 等，Tomcat 就使用 Debian 作为操作系统层。每一层都是在前一层的基础上镜像修改和提交的结果。

不同层之间并不是孤立存在的，而是有着非常强的依赖关系，上层必须依赖下层存在。因为 Docker 镜像中的层级都可以理解为一个单独的镜像，而这个镜像其实是由下层的镜像进行修改得来的，这层镜像只记录了对前一层镜像的修改，并非包含了前一层镜像的所有内容，这也是与 Git 的工作原理不同的地方。正是因为镜像的层级依赖机制，才使得 Docker 镜像更容易被拆分并镜像共享，凸显了镜像占用空间小、容易扩展、传输灵活等优点。

### 2.1.4 镜像的写时复制

Docker 里的镜像技术还有一个特性——写时复制。在一些编程语言中，写时复制是指在复制某个数组或对象时，复制的过程并不是马上发生，而是先进行一些标记动作，只有需要对复制的数组或对象进行修改时，才会真正复制出这个变量的副本。Docker 镜像的写时复制机制与之相似。

通过镜像在 Docker 中运行容器时，并不是马上就把镜像的所有内容复制到沙盒环境



下，只是直接把沙盒环境建立在镜像的基础上。容器运行的沙盒环境其实就是镜像之上的一层新的可读写的镜像层，原有的镜像以只读的方式被衔接在新镜像层的下方。也就是说，容器的启动并不需要任何复制，也不需要单独开辟硬盘空间，只有容器中的程序对文件进行修改时，才会将原有镜像中这个需要被修改的文件复制到沙盒环境的镜像层中。

Docker 的写时复制机制既保证了原有镜像不会受到容器运行的影响，又通过消除容器启动时复制镜像到沙盒环境的过程，大幅提升了容器的启动速度。可以说，Docker 容器能够达到秒级的启动速度，写时复制机制在其中发挥了举足轻重的作用。

## 2.2 使用和管理镜像

Docker 提供了丰富的管理镜像的方法，我们可以通过简单的指令对 Docker 镜像进行构建、列表、查询信息、删除等操作，也能够利用这些方法很轻松地将镜像共享到互联网上或者私有的镜像仓库中。下面我们就向大家介绍常见的 Docker 镜像管理命令。

### 2.2.1 获取镜像

镜像是容器的基础，在运行容器之前，必须获得容器的镜像。在 Docker CLI 中，我们可以通过 `docker pull` 命令从互联网上下载需要的镜像。例如，拉取一个 Ubuntu 系统的镜像到本地镜像库中：

```
$ sudo docker pull ubuntu
Using default tag: latest
latest: Pulling from library/ubuntu

43db9dbdcb30: Downloading 4.56 MB/49.33 MB
85a9cd1fcca2: Download complete
c23af8496102: Download complete
e88c36ca55d8: Download complete
```

我们在下载 Ubuntu 系统时，没有指定镜像的命名空间，因为我们需要的 Ubuntu 镜像属于 Docker 官方管理的镜像，这些镜像是没有命名空间的。同时，我们也没有指定镜像的标签，这个时候 Docker 会自动使用 `latest` 这个默认的标签。通常情况下，我们倾向于使用更确定的镜像信息，减少镜像版本带来的不确定性：



```
$ sudo docker pull openresty/openresty:1.9.15.1-centos
1.9.15.1-centos: Pulling from openresty/openresty

a3ed95caeb02: Pull complete
da71393503ec: Downloading 2.162 MB
c96d59514c43: Downloading 2.153 MB/64.57 MB
```

Docker 会利用镜像的分层机制，将镜像分为多个包进行下载，我们可以在终端输出中看到每层的下载状态。当所有镜像层的下载和解压等操作完成，它就会出现在本地的镜像仓库中，与此同时，终端屏幕上会输出镜像下载成功的提示：

```
Digest: sha256:7ce82491d6e35d3aa7458a56e470a821baecee651fba76957111402591d20fc1
Status: Downloaded newer image for ubuntu:latest
```

互联网上已经存在很多由组织或个人提供的镜像，我们可以通过 `docker search` 命令在 Docker 官方提供的镜像共享仓库 Docker Hub 中查找需要的镜像：

```
$ docker search php
```

NAME	DESCRIPTION	STARS	OFFICIAL	AUTOMATED
php	While designed for web...	1360	[OK]	
richarvey/nginx-php-fpm	Container running Nginx ...	232		[OK]
php-zendserver	Zend Server - the integrated...	77	[OK]	
eboraas/apache-php	PHP5 on Apache (with SSL...	75		[OK]
million12/nginx-php	Nginx + PHP-FPM ...	73		[OK]

屏幕显示出来的搜索结果中包含了镜像的主要信息。

- ☐ NAME: 名称。
- ☐ DESCRIPTION: 创建者提供的对镜像的简单描述。
- ☐ STARS: 镜像在官方镜像仓库中收到用户给出的星星的数量，表示镜像受欢迎的程度。
- ☐ OFFICIAL: 镜像是否由 Docker 官方提供。我们更倾向于使用官方提供的镜像，因为它们更稳定。
- ☐ AUTOMATED: 镜像是否使用了自动构建。

通过 `docker search` 和 `docker pull` 命令可以很轻松地从网上下载所需要的镜像，当然，对于已经可以使用互联网获取到的镜像，我们推荐直接下载使用。

## 2.2.2 列出镜像

在使用 Docker 时,经常需要确定在本地镜像库中都有哪些镜像,以确保在使用的時候可以直接调用而不用等待网络加载。如果想要查看本地镜像库的镜像列表,可以使用列出镜像命令:

```
$ sudo docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
ubuntu	latest	ac526a356ca4	4 days ago	125.2 MB
php	7-fpm	75b880f3a420	9 days ago	375 MB
alpine	3.4	4e38e38c8ce0	4 weeks ago	4.799 MB
nginx	1.10	82e97a2b0390	7 weeks ago	182.8 MB
php	5.6-fpm	818f51a5c05c	8 weeks ago	455.5 MB
mysql	5.6	2c0964ec182a	8 weeks ago	329 MB

命令所返回的镜像列表主要包含了以下 5 个字段。

- ❑ **REPOSITORY**: 镜像的名称,如果镜像命名中含有 Namespace,这里也会一并显示。在某些场合下,镜像的名称会显示为<none>,比如创建时没有指定名称的时候,或者新建镜像时使用了和之前镜像相同的名称的时候。
- ❑ **TAG**: 镜像的标签。
- ❑ **IMAGE ID**: 镜像的 ID。镜像的 ID 是更精确地识别镜像的属性,它由 SHA256 散列算法生成,表示为一个长度为 64 位的十六进制字符串。为了节省页面,这里只显示了镜像 ID 的前部分,不过这也足以说明问题。
- ❑ **CREATE**: 镜像的创建时间。
- ❑ **SIZE**: 镜像所占用的硬盘空间,包括被共享的镜像层的大小。

当本地镜像较多时,还可以使用通配符过滤出符合条件的镜像:

```
$ sudo docker images ph*
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
php	7-fpm	75b880f3a420	10 days ago	375 MB
php	5.6-fpm	818f51a5c05c	8 weeks ago	455.5 MB

## 2.2.3 获得镜像的详细信息

除了能在镜像列表中获取信息,还可以通过 `docker inspect` 命令获得镜像更详细的信息:

```

$ sudo docker inspect nginx:1.10
[
  {
    "Id": "sha256:82e97a2b0390a20107ab1310dea17f539ff6034438099384998fd91fc540b128",
    "RepoTags": [
      "nginx:1.10"
    ],
    "RepoDigests": [],
    "Parent": "",
    "Comment": "",
    "Created": "2016-06-01T18:03:14.295411233Z",
    "Container": "59729278b12c491183cc03db215bc75ff3654e3fe349a90e84be7884bcc89f0e",
    "ContainerConfig": {
      "Hostname": "b0cf605c7757",
      "Domainname": "",
      "User": "",
      "AttachStdin": false,
      "AttachStdout": false,
      "AttachStderr": false,
      "ExposedPorts": {
        "443/tcp": {},
        "80/tcp": {}
      },
      "Tty": false,
      "OpenStdin": false,
      "StdinOnce": false,
      "Env": [
        "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
        "NGINX_VERSION=1.10.1-1~jessie"
      ],
      "Cmd": [
        "/bin/sh",
        "-c",
        "#(nop) CMD [\"nginx\" \"-g\" \"daemon off;\"]"
      ],
      "Image": "caef6cc4000d0ee57218fd791bbf037f018c5cf4d88a1372dcc7de98275779d7",
      "Volumes": null,
      "WorkingDir": "",
      "Entrypoint": null,
      "OnBuild": [],
      "Labels": {}
    }
  ],
]

```



```
"DockerVersion": "1.9.1",
"Author": "NGINX Docker Maintainers \"docker-maint@nginx.com\"",
"Config": {
  "Hostname": "b0cf605c7757",
  "Domainname": "",
  "User": "",
  "AttachStdin": false,
  "AttachStdout": false,
  "AttachStderr": false,
  "ExposedPorts": {
    "443/tcp": {},
    "80/tcp": {}
  },
  "Tty": false,
  "OpenStdin": false,
  "StdinOnce": false,
  "Env": [
    "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
    "NGINX_VERSION=1.10.1-1~jessie"
  ],
  "Cmd": [
    "nginx",
    "-g",
    "daemon off;"
  ],
  "Image": "caef6cc4000d0ee57218fd791bbf037f018c5cf4d88a1372dcc7de98275779d7",
  "Volumes": null,
  "WorkingDir": "",
  "Entrypoint": null,
  "OnBuild": [],
  "Labels": {}
},
"Architecture": "amd64",
"Os": "linux",
"Size": 182770326,
"VirtualSize": 182770326,
"GraphDriver": {
  "Name": "aufs",
  "Data": null
},
"RootFS": {
  "Type": "layers",
```



```

    "Layers": [
      "sha256:4dcab49015d47e8f300ec33400a02cebc7b54cadd09c37e49eccbc655279da90",
      "sha256:5f70bf18a086007016e948b04aed3b82103a36bea41755b6cddfaf10ace3c6ef",
      "sha256:5f70bf18a086007016e948b04aed3b82103a36bea41755b6cddfaf10ace3c6ef",
      "sha256:5f70bf18a086007016e948b04aed3b82103a36bea41755b6cddfaf10ace3c6ef",
      "sha256:5f70bf18a086007016e948b04aed3b82103a36bea41755b6cddfaf10ace3c6ef",
      "sha256:eedb9dd672413c3435398a1263ebb1f80734ec89a35b3b7658445d7ca83745a0",
      "sha256:7a96d76c274356e82b14a9e6d315e7025a29cc7f4e87e2da15310885b6c52df1",
      "sha256:5f70bf18a086007016e948b04aed3b82103a36bea41755b6cddfaf10ace3c6ef",
      "sha256:5f70bf18a086007016e948b04aed3b82103a36bea41755b6cddfaf10ace3c6ef"
    ]
  }
}
]

```

`docker inspect` 命令可以获得镜像的详细信息，也可以接收镜像的名称、镜像的 ID 等唯一性信息：

```
$ sudo docker inspect 82e97a2b0390
```

Docker 允许在命令中只给出部分的 ID，这里我们只使用了镜像 ID 的前 12 位，不过这足以识别镜像在本地的唯一性了，也可以得出我们想要的结果。当然，我们还可以只提供镜像 ID 的前 8 位、前 5 位，甚至前两位，只要我们能够保证给出的前缀可以在本地找到唯一对应的镜像即可。

使用 `docker inspect` 命令会以 json 格式返回镜像的很多信息，如果只想得到某部分信息，可以使用输出格式化进行过滤：

```
$ sudo docker inspect -f {{".Size"}} 82e97a
182770326
```

## 2.2.4 删除镜像

如果需要删除本地镜像，可以使用 `docker rmi` 命令：

```

$ sudo docker rmi php:5.6-fpm
Untagged: php:5.6-fpm
Deleted: sha256:818f51a5c05cf39c88ec7e8276e16510bb09714ef8526e8ff66ecd8d1d1307d4
Deleted: sha256:3784571a6dfd72c93a880f9919157cb8dea9e59acdfa0bce0d5fc3a735c9c6
Deleted: sha256:ea165e58caeac30f25e054bea35be32742988df51f0ba65c9164b9858b91f1ea
Deleted: sha256:5a07d483753e6731e6bacb5b9f9fc6f4f757f8c3b3ce562cc27057b107a69a6
Deleted: sha256:3f8534c8f2901f6bc49e2d392f188de3bd0392d3b0d6e7bd1f60d487eb2467b4
Deleted: sha256:c6da4711e3d0c92c72816a8855d6a8038006e08412125f20708594d5062a4878
Deleted: sha256:6b9ce12f069ed802bd12236971ab6990c34d9d51cb4eb93a0ce4651c4b372c78

```

```
Deleted: sha256:0eebe2ee207943168d95d4ad87b5e7ea73afe4a62aea4521dbb871bfab4b708e
Deleted: sha256:6929110f144a30fad22df15f54729ca560a666f6458427f4aafefbc594ae77b25
Deleted: sha256:544a6bd322577416287c226a63042888779c4697c30120f0c67904ed855d91da
Deleted: sha256:e2d5237e993685235ca99bd2cba4827aa7b0b4a1be9cf88c509de1393b2f7dd8
Deleted: sha256:74a267d47c8dd9a0dd958f03256221840b24c5a774abe99568997cb56bab390c
Deleted: sha256:1cd4e1e639b409830ca890e412c1454161ffec4f1a975fd8f578ad4a35519e74
Deleted: sha256:b70ea3181dc6e8d1ce4d2c5c59bcd41ef91e90da81e700f210439b7a032cab
Deleted: sha256:861ebe92c25de30c50e8fcd3316c6aa23f052a5455fe35816c661e4b4df5f114
```

如果一个镜像中含有某些与其他镜像共享的镜像层，这些镜像层仍然会保留下来，只有未被其他镜像使用的层会被删除。

我们同样可以指定镜像 ID 来删除镜像：

```
$ sudo docker rmi 4e38e38c8ce0
Untagged: alpine:3.4
Deleted: sha256:4e38e38c8ce0b8d9041a9c4fefe786631d1416225e13b0bfe8cfa2321aec4bba
Deleted: sha256:4fe15f8d0ae69e169824f25f1d4da3015a48feeeeabb265cd2e328e15c6a869f
```

镜像是要被容器使用的，如果我们删除镜像时正有容器在使用它，那它将无法被删除。我们可以使用携带 `-f` 参数的 `docker rmi` 命令强制删除镜像：

```
$ sudo docker rmi -f 4e38e38c8ce0
```

不过并不推荐使用这种方法，因为这会带来一些难以处理的遗留问题，所以最好还是先关闭或清理掉使用此镜像的容器再删除镜像。

## 2.2.5 镜像的迁移

虚拟化部署的高效率，主要是镜像无缝迁移的功劳，也就是说，镜像的导入/导出是虚拟化过程中最基础的操作，Docker 也提供了镜像导出和载入本地镜像库的方法。

使用 `docker save` 命令可将本地镜像库中的镜像导出：

```
$ sudo docker images
REPOSITORY          TAG             IMAGE ID        CREATED         SIZE
ubuntu              latest         ac526a356ca4   7 days ago     125.2 MB
...
$ sudo docker save -o ubuntu.tar ubuntu:latest
```

默认情况下，`docker save` 命令是把镜像写出到输出流中，可以通过 `-o` 或 `--output` 参数将导出的镜像数据写入到指定文件，也可以使用 `>` 写出到指定文件：

```
$ sudo docker save ubuntu:latest >ubuntu.tar
```

另外，`docker save` 命令支持同时导出多个镜像，只要将镜像逐个传入即可：

```
$ sudo docker save -o images.tar ubuntu:latest centos:latest
```

要将导出的镜像数据重新导入到本地的镜像仓库中，可以使用 `docker load` 命令：

```
$ sudo docker load -i ubuntu.tar
```

默认情况下，`docker load` 命令从输入流中读取镜像数据，可以通过 `-i` 或 `--input` 参数将指定的文件传入镜像数据，也可以通过 `<` 来完成：

```
$ sudo docker load <ubuntu.tar
```

## 2.3 Docker Hub 简介

为了让世界各地使用 Docker 的工程师们能够更好地共享彼此的镜像，Docker 提供了部署在互联网服务器上的镜像仓库。而 Docker Hub 就是由 Docker 官方提供的镜像仓库，也是目前最权威、最庞大的 Docker 镜像仓库。那么，镜像仓库究竟为何物？Docker Hub 应该怎样使用，它能为我们带来怎样的便利？这些问题我们将在本节进行解读。

### 2.3.1 镜像仓库

我们提到了一个词——镜像仓库，那么何为镜像仓库，它能给我们带来怎样的便利呢？Docker 在镜像模块的结构设计上，借鉴了很多 Git 代码版本控制的概念，镜像仓库就是其中之一。和代码仓库用于集中存放不同版本和分支的代码的作用类似，镜像仓库用于集中存放 Docker 镜像。

镜像仓库分为本地镜像仓库和远程镜像仓库。本地镜像仓库主要支持我们对镜像的操作并为容器的运行提供镜像，远程镜像仓库更多地用于对镜像的分发。前面我们提到了通过镜像的导入/导出来迁移镜像，但是这种方式仍然比较烦琐并容易出现纰漏，而镜像仓库则是迁移镜像更好的一种方式。通过分布在世界各地的镜像仓库，我们可以更好地利用互联网进行镜像的共享和分发。图 2-5 展示了镜像仓库与镜像之间的关系。

镜像名称在镜像仓库中代表唯一的镜像。因为同一镜像仓库中的镜像很可能由很多用户共同提交和维护，他们所提供支持的镜像仓库名称也很容易出现重名的情况，所以在镜像仓库中需要通过命名空间来区别不同用户上传和维护的镜像。



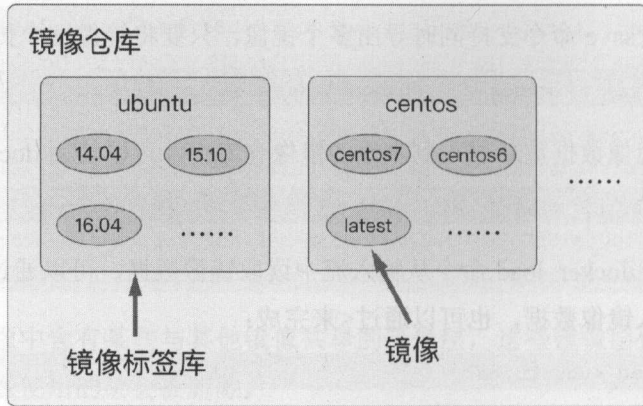


图 2-5 镜像仓库的组织结构

镜像仓库主要囊括了镜像管理系统和用户系统。镜像管理系统为镜像仓库提供了类似代码库式的镜像存取和管理方式，而用户系统则为镜像仓库中的镜像管理操作的授权提供支持。

如果远程镜像仓库需要进行用户登录和授权操作，可以使用 `docker login` 命令登录到指定的镜像仓库服务器：

```
$ sudo docker login -u <username> -p <password> <server>
```

使用 `docker login` 命令时，可以通过 `-u` 或 `--username` 参数指定用户名，通过 `-p` 或 `--password` 参数指定登录密码，在最后还可以给出需要连接的镜像仓库服务器的地址。如果没有给出镜像仓库服务器的地址，会采用 Docker Daemon 中的默认值。

由于用户名和密码都是敏感信息，所以我们不推荐在 `docker login` 的参数中将其携带传入，而是输入命令后由程序给出单独的输入流进行输入：

```
$ sudo docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't have
a Docker ID, head over to https://hub.docker.com to create one.
Username: ...
Password:
Login Succeeded
```

登录成功时，我们会收到 `Login Succeeded` 这条消息。

## 2.3.2 Docker Hub 简介

由世界各地的组织或个人提供的镜像仓库有很多，其中最具影响力的当属 Docker Hub。Docker Hub 是 Docker 官方提供的镜像仓库，也是全球最大的镜像仓库。



Docker Hub 的网址是 <https://hub.docker.com/>，除了提供普通镜像仓库所包含的功能，还提供了搜索、创建、分发和管理等图形界面操作的支持。另外，Docker Hub 还提供了云端镜像构建服务，让制作镜像的过程在支持自动化构建的集群服务器中完成，有效节约了本地主机的资源消耗和构建镜像耗费的时间。因为 Docker 容器技术全方位地保证了基于镜像创建的容器在不同主机上运行的一致性，所以无须担心在远程构建的镜像下载到本地时会出现“水土不服”的情况。

Docker Hub 为所有用户提供不限量的公开镜像托管服务，不过只提供了一个免费的私有镜像托管名额，如果需要增加私有镜像的数量，需要单独付费。

托管在 Docker Hub 中的镜像主要分为两类：一类是由官方提供，交由可靠、权威的第三方组织或机构维护的官方镜像；另一类就是普通用户提供的镜像。官方镜像最大的特点就是其镜像仓库名称前是没有命名空间的。

### 2.3.3 注册 Docker Hub 账号

要使用 Docker Hub 的服务，首先需要注册一个 Docker Hub 账号。如图 2-6 所示，当我们在未登录的情况下进入到 Docker Hub 的主页（<https://hub.docker.com/>）时，便能看到创建账号的提示和 Docker Hub ID、注册邮箱及密码的输入框。如果已经注册了 Docker Hub 账号，可以直接单击右上角的 Log In 登录。

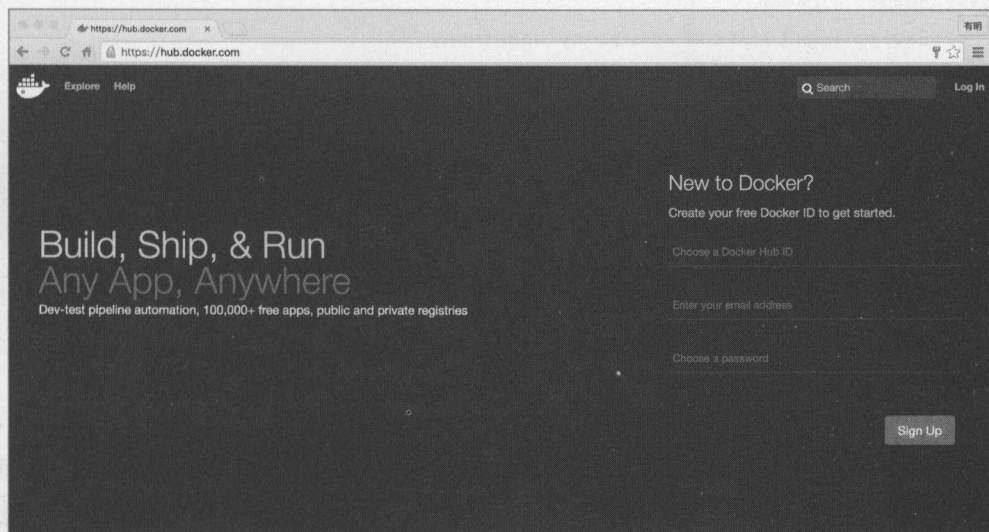


图 2-6 Docker Hub 首页

如图 2-7 所示，进入到 Docker Hub 中，就能看到托管在 Docker Hub 的镜像了。我

们可以在本地使用 `docker login` 登录到 Docker Hub 中，再使用 `docker push` 将镜像发布到 Docker Hub 上。

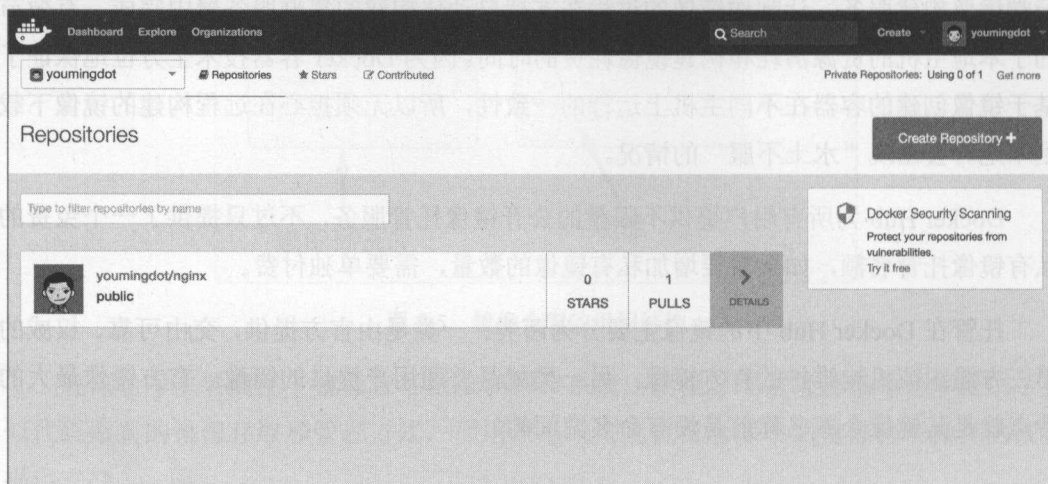


图 2-7 Docker Hub 中的个人页面

**小提示：**使用 `docker login` 命令时不添加 `<server>`，默认使用的就是 Docker Hub。

`docker push` 是与 `docker pull` 相对的一个命令，其使用方法与 `docker pull` 类似：

```
$ sudo docker push youmingdot/nginx:latest
82640e8ba1b0: Pushed
0ec46088e6ca: Pushed
5f70bf18a086: Mounted from openresty/openresty
7a96d76c2743: Mounted from library/nginx
eedb9dd67241: Mounted from library/nginx
4dcab49015d4: Mounted from library/mysql
latest: digest: sha256:9ad5160266be6a4d1536a668b03c47ff5416b4d41c67b9a7520f717
fff962368 size: 2393
```

与 `docker pull` 不同的是，使用 `docker push` 时需要保证对远程镜像仓库有写入的权限。

## 2.3.4 搜索镜像

Docker Hub 汇集了世界各地的开发者们提供的镜像，要使用好 Docker Hub，必须学会如何使用 Docker Hub 的搜索功能。在 Docker Hub 页面的头部，可以看到用于搜索镜像的搜索框，只要将需要检索的关键字输入其中并按回车键，就能得到 Docker Hub 中相关的镜像结果。图 2-8 展示了搜索 `tomcat` 镜像的结果。

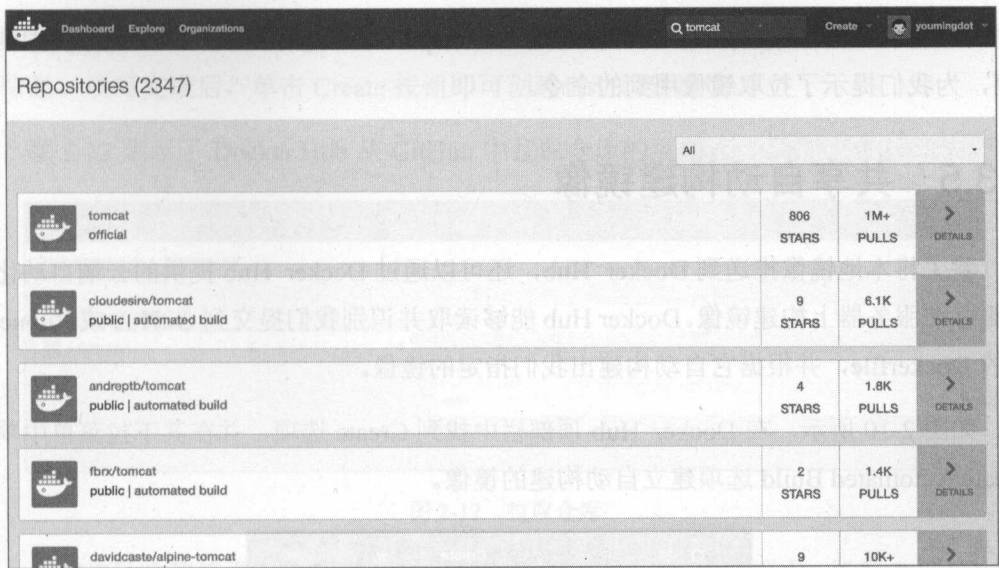


图 2-8 搜索 tomcat 镜像的结果

如图 2-9 所示，在搜索结果中，我们能够看到与使用 `docker search` 命令得到的结果相似的属性字段（包括镜像的名称、被拉取次数、评星数）是否为官方镜像等。

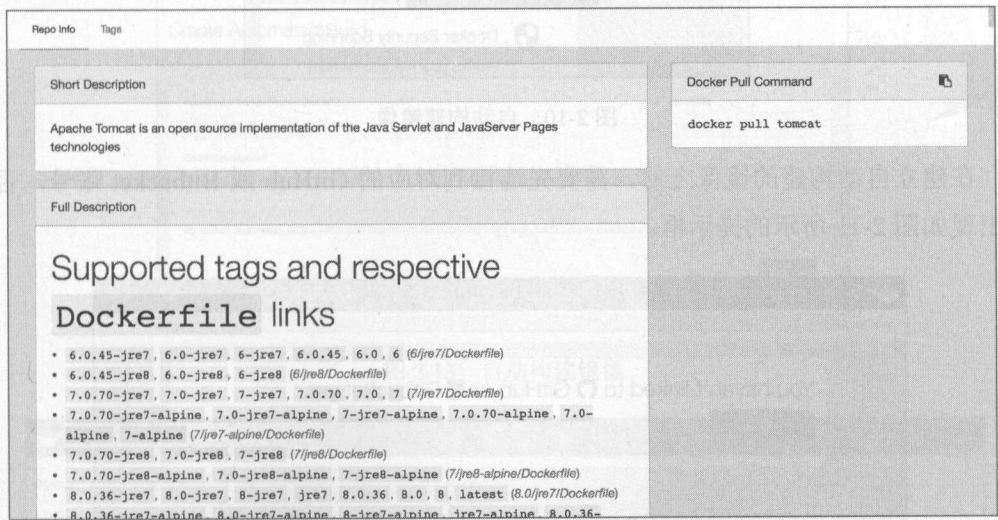


图 2-9 属性字段

进入镜像页面，可以看到镜像更详细的信息，包括镜像提供者对镜像撰写的简介及详细的介绍。在详细介绍之前，会展示镜像所有可用的标签，以及构建这些标签所使用的 Dockerfile 所在的代码库链接。Dockerfile 是 Docker 中用于自动化构建镜像的配置文件，我们会在之后的章节中专门进行讲解。在详细介绍中，镜像的提供者通常会介绍镜像的使用方法。



找到我们需要的镜像后，就可以通过 `docker pull` 命令进行拉取了，在镜像页面的右侧，为我们提示了拉取镜像用到的命令。

### 2.3.5 共享自动构建镜像

除了将本地镜像推送到 Docker Hub，还可以通过 Docker Hub 提供的云端自动化构建服务在服务器上构建镜像。Docker Hub 能够读取并识别我们提交到 GitHub 或 Bitbucket 上的 Dockerfile，并根据它自动构建出我们指定的镜像。

如图 2-10 所示，在 Docker Hub 顶部栏中找到 Create 选项，并在其下拉菜单中单击 Create Automated Build 选项建立自动构建的镜像。

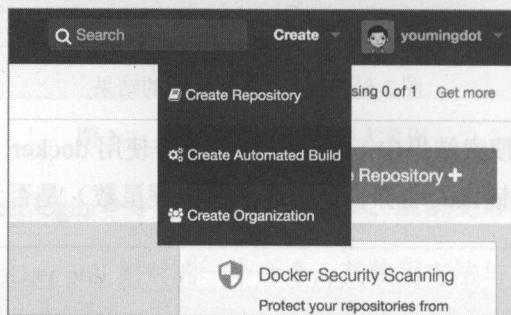


图 2-10 自动构建镜像

在建立自动构建的镜像之前，需要先连接到对应的 GitHub 或 Bitbucket 账号，否则会出现如图 2-11 所示的提示框。

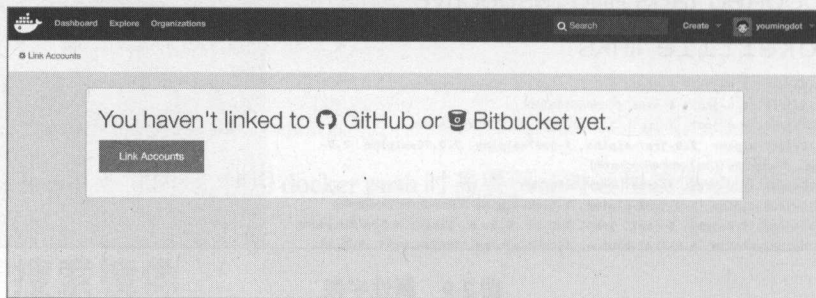


图 2-11 提示框

连接到 GitHub 或 Bitbucket 后，就可以选择带有 Dockerfile 的代码仓库并将其作为构建所使用的仓库。选择完自动构建所需要配置文件所在的网络代码仓库后，Docker Hub 服务器会对自动构建配置所在的代码仓库镜像扫描，查找出其中用于配置的 Dockerfile 文件，并使用它进行自动构建。



与此同时，我们还要为即将构建的镜像配置镜像仓库名、访问权限及撰写镜像的描述信息。填写完成后，单击 **Create** 按钮即可创建自动构建镜像。

图 2-12 显示了 Docker Hub 从 GitHub 中拉取仓库的页面。

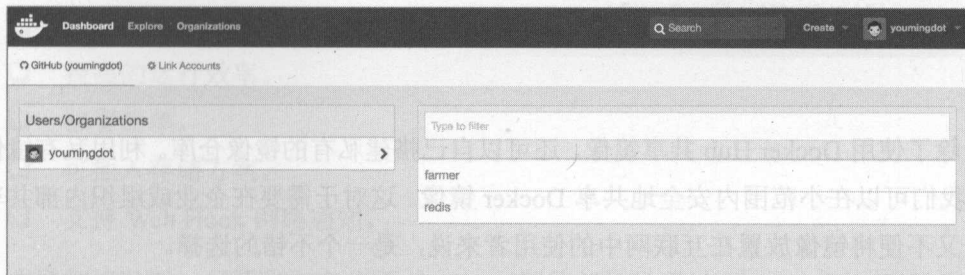


图 2-12 拉取仓库

完成自动构建镜像的镜像创建后，如图 2-13 所示，进入镜像页面，找到 **Build Details** 页面，就能够看到镜像构建的状态。当构建状态变为 **Success** 时，就表示构建镜像的过程已经完成。图 2-14 显示了镜像构建完成的状态。

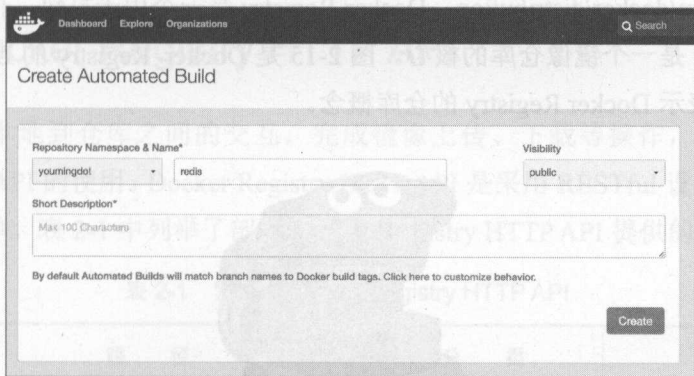


图 2-13 自动构建镜像

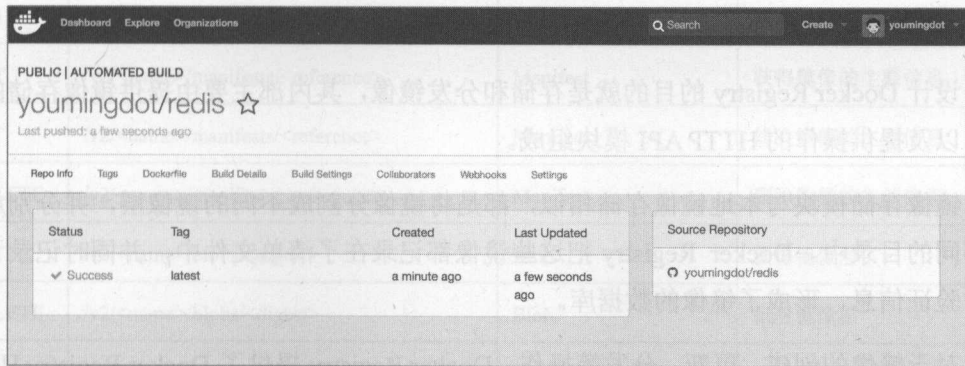


图 2-14 镜像构建完成

使用 Docker Hub 提供的自动构建服务，不但能够节约本地主机的资源，而且能很好地利用 Docker Hub 提供的服务器高性能的特点，快速完成镜像的构建。

## 2.4 搭建私有仓库

除了使用 Docker Hub 共享镜像，还可以自己搭建私有的镜像仓库。利用私有镜像仓库，我们可以在小范围内安全地共享 Docker 镜像。这对于需要在企业或组织内部共享镜像，又不便将镜像放置在互联网中的使用者来说，是一个不错的选择。

### 2.4.1 镜像分发服务

除了使用 Docker Hub 分发镜像，Docker 还提供了开源的镜像分发工具——Docker Registry。目前，Docker Registry 已经更新到了 2.0 版本，开源在 GitHub 上，网址为 <https://github.com/docker/distribution>。Docker Registry 是一个用于打包、传输、存储和分发的工具集合，是一个镜像仓库的核心。图 2-15 是 Docker Registry 的 Logo，以扇贝集装箱的形象表示 Docker Registry 的仓库概念。

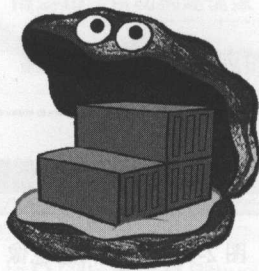


图 2-15 Docker Registry

设计 Docker Registry 的目的就是存储和分发镜像，其内部主要由提供镜像存储的模块，以及提供操作的 HTTP API 模块组成。

镜像存储模块与本地镜像存储相似，都是将镜像分割成不同的镜像层，并分别放置在不同的目录中。Docker Registry 把这些镜像都记录在了清单文件中，并同时记录了它们的验证信息，形成了镜像的数据库。

对于镜像的创建、更新、分发等操作，Docker Registry 提供了 Docker Registry HTTP

API 系列接口来实现对操作的调用。通过 Docker Registry HTTP API，我们可以从远程实现对 Docker Registry 中镜像数据的交互。

Docker Registry 在不断的迭代、优化中，积累了很多优点：

- ☐ 高速的上传和下载镜像。
- ☐ 极高的运行效率。
- ☐ 部署简单。
- ☐ 可插入存储方案。
- ☐ 支持 Web Hook 网络通知。

在通常情况下，只要我们安装了 Docker，就能轻松地通过 Docker Hub 下载我们需要的镜像，如果我们拥有 Docker Hub 的账号，还能够向 Docker Hub 提交我们的公开镜像。这些功能对于部分用户、组织或企业来说是足够的，但是却不能满足某些人的需求。例如，某些企业需要存储含有自己的应用程序的镜像，或者需要存放用于测试和持续继承的镜像，此时拥有一个私有的镜像仓库是更佳的选择。

## 2.4.2 Docker Registry HTTP API

要实现从本地到仓库之间的交互，完成镜像上传、下载等操作，离不开 Docker Registry HTTP API 的使用。Docker Registry HTTP API 是采用 RESTful 设计的接口，使用的方法非常简单。表 2-1 中列举了部分 Docker Registry HTTP API 提供的接口。

表 2-1 常见的Docker Registry HTTP API

方法	路 径	分 类	简 介
GET	/v2/	Base	检查是否支持2.0接口
GET	/v2/<name>/tags/list	Tags	获得镜像的标签列表
GET	/v2/<name>/manifests/<reference>	Manifest	获得镜像的主要信息
PUT	/v2/<name>/manifests/<reference>	Manifest	修改镜像的主要信息
DELETE	/v2/<name>/manifests/<reference>	Manifest	删除镜像的主要信息
GET	/v2/<name>/blobs/<digest>	Blob	获得镜像层
DELETE	/v2/<name>/blobs/<digest>	Blob	删除镜像层
POST	/v2/<name>/blobs/uploads/	Initiate Blob Upload	开始分块上传



续表

方法	路 径	分 类	简 介
GET	/v2/<name>/blobs/uploads/<uuid>	Blob Upload	获得分块上传进度
PATCH	/v2/<name>/blobs/uploads/<uuid>	Blob Upload	分块上传数据
PUT	/v2/<name>/blobs/uploads/<uuid>	Blob Upload	完成上传
DELETE	/v2/<name>/blobs/uploads/<uuid>	Blob Upload	取消上传
GET	/v2/catalog	Catalog	获得镜像列表

使用 Docker Registry HTTP API 时，接触最多的就是镜像传输块（Blob）和镜像的主要信息（Manifest）了。

镜像的主要信息以 json 形式展示，记录了镜像的名称、标签、镜像层等信息。

```
{
  "name": <name>,
  "tag": <tag>,
  "fsLayers": [
    {
      "blobSum": <digest>
    },
    ...
  ],
  "history": <v1 images>,
  "signature": <JWS>
}
```

每个镜像层都存在一个 digest，我们称之为内容摘要（Content Digest），用于镜像内容寻址。Docker Registry 采用内容寻址存储（CAS）方法，根据数据单元的唯一 ID 与元数据（Metadata）一起构成访问数据的实际地址，从而保证数据存储的稳固。内容摘要就是根据内容寻址中的访问地址，产生出的一串哈希校验码，因为其以哈希摘要的形式存在，所以在传输过程中可以确保存储信息不会被泄露。

2.4.3 部署私有仓库

Docker 提供的镜像仓库服务是运行在 Docker 容器中的，我们可以通过下面的命令很方便地搭建并运行镜像仓库服务：



```
$ sudo docker run -d --name private-registry -hostname localhost \
-v /opt/docker/distribution:/var/lib/registry/docker/registry-v2 \
-p 5000:5000 registry:2.0
```

若运行容器的基础镜像，我们可以通过 Docker 开源在 GitHub 上的代码 (<https://github.com/docker/distribution>) 进行编译。

当 Docker Registry 服务运行起来后，我们就能够向它推送和拉取镜像了：

```
$ sudo docker push localhost:5000/youmingdot/redis:latest
```

在实际生产中，直接暴露 Docker Registry 的 5000 端口是不安全的，我们可以通过反向代理的方法让 Docker Registry 使用 HTTPS 安全协议，并且可以加上一些基础认证，让 Docker Registry 更安全。

我们可以通过 Nginx 实现反向代理的搭建，在 Nginx 中开启 HTTPS 安全协议，使用户在连接和使用 Docker Registry 时通过 Nginx 在 TLS 安全传输层上进行数据的传输，而 Docker Registry 本身不对外暴露 5000 端口，只是从 Nginx 中获取到用户发送来的操作指令。

在 Nginx 中配置带有 HTTPS 的代理其实非常简单，首先通过 openssl 工具生成一份密钥和证书：

```
$ openssl genrsa -out https.key 4096
$ openssl req -newkey rsa:4096 -nodes -keyout https.key -x509 -days 3650 \
-out https.crt -subj "/C=CN/ST=zj/L=hz/O=yymd/OU=localhost"
```

再在 Nginx 的主机配置中加入 HTTPS 配置：

```
server {
    ssl on;
    ssl_certificate ~/https.crt;
    ssl_certificate_key ~/https.key;
    ...
}
```

然后在配置中加入反向代理的配置：

```
server {
    ...
    location / {
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-By $server_addr:$server_port;
```

```
proxy_set_header X-Forwarded-For $remote_addr;

proxy_pass http://registry:5000;
}
}
```

之后重启 Nginx 服务，当 Nginx 再次运行起来时，我们就可以通过 HTTPS 协议来访问 Docker Registry 了。

既然已经使用了 Docker 运行 Docker Registry，那我们也推荐选择使用 Docker 来搭建 Nginx 服务器。关于如何通过 Docker 搭建 Nginx 服务器，如何配置外部网络、Nginx、Docker Registry 之间的通信，如何确保 Nginx 容器和 Docker Registry 容器之间的安全隔离性等问题，我们将在之后的章节中逐步介绍。

## 2.5 本章小结

在本章中，我们了解了 Docker 镜像的基本知识，熟悉了镜像的下载、删除、导入、导出等操作；介绍了镜像仓库的知识，了解了如何通过镜像仓库进行镜像的获取、传输、分发等操作；简单讲解了官方镜像仓库 Docker Hub 的基本使用方法，以及如何搭建一个私有的镜像仓库。

镜像作为 Docker 容器运行的基础，是 Docker 组成中不可或缺的一部分，掌握镜像与镜像仓库的知识，可以帮助我们更好地使用 Docker。

# 第 3 章

## 管理和使用容器

在 Docker 中，容器是基于镜像运行的轻量级环境，是 Docker 封装和管理应用程序或微服务的“集装箱”。运行中的容器读取了镜像中基础的程序和依赖库的代码，并将修改保存在一个沙盒环境中，充分保障了应用程序运行的虚拟性和隔离性。

容器作为 Docker 的核心组件，熟悉掌握容器的操作是使用 Docker 不可或缺的技能，在本章中，我们就专门介绍 Docker 容器的使用。我们将了解到容器的创建、终止、删除等基本操作，以及如何查看运行在容器中的应用程序的状态，如何导入、导出和迁移容器。

### 3.1 管理容器

要掌握容器的使用方法，就要了解管理 Docker 容器的基本方法。作为容器技术的核心模块，Docker 为我们提供了非常丰富的对容器的操作，这些操作足以让我们创建、启动、停止和了解容器的运行状况。

#### 3.1.1 创建容器

所有的容器操作都离不开容器，容器是在镜像的基础上建立的运行态，要创建容器，可以使用 `docker create` 命令。创建容器的命令非常简单，将容器所基于的镜像名称传入



即可。Docker 会从本地镜像仓库中找到给定的镜像，如果未找到本地镜像，也会自动从远程镜像仓库中下载需要的镜像。这里，我们创建一个运行 Debian 系统的容器：

```
$ sudo docker create debian:jessie  
2e4ee3689de6f047008af45e95f1993688e5a16e500c55201a1a8d0d
```

当容器创建完成后，Docker 会马上将容器的 ID 输出到屏幕上。容器的 ID 与镜像 ID 类似，是可以识别容器唯一性的属性，显示为长度为 64 位的十六进制字符串。每个容器的 ID 都是独一无二的，即使它们基于同一个镜像创建。

容器的 ID 能够识别容器的唯一性，这在需要指定容器的时候显得非常重要。在很多场合下，我们也能像在镜像操作中给出部分镜像 ID 一样，使用容器 ID 的前一部分，只要我们能够确保给出的部分 ID 在本地主机所有的容器中对应唯一的容器即可。

Docker 容器是针对运行单一的应用程序设计的，对于容器生命周期的控制，Docker 通过将其与进程的生命周期进行绑定来实现。虽然 Docker 容器中能够运行多个进程乃至多个不同的程序，但在设计上应该只以一个应用程序为主体，其他程序只是为这个主体提供支持。容器启动时，一个被我们指定的程序也会被启动，Docker 会监视这个程序的主进程，当这个进程退出时，容器也会停止运行。而当我们通过命令来停止容器运行时，Docker 会发送停止信号给这个进程，让程序结束。

在大多数镜像中，镜像的制作者已经给定了基于这个镜像的容器在启动时所要运行的程序，也可以在创建容器的同时重新指定容器所绑定的应用程序。通过在指定进程名称的后面携带程序名称和参数，就能实现重新绑定主应用程序了：

```
$ sudo docker create debian:jessie tail -f /var/log/faillog
```

需要注意的是，Docker 容器只会绑定一个进程，所以我们重新指定的进程会覆盖镜像原本设置的应用程序绑定。如果应用程序依赖于其他程序的支持，而这些程序也需要放置和运行在同一个容器中，我们可以通过容器入口来运行这些程序。

了解了容器的创建，接下来要了解的就是运行容器。需要注意的是，Docker 和其他虚拟机类似，创建容器只是配置了容器的文件系统和网络等基础资源，并不代表容器及容器中的应用程序已经在运行了。

若要在创建容器的同时就让容器运行起来，需要使用 `docker run` 命令，其与 `docker create` 命令很相似。

```
$ sudo docker run debian:jessie
```

Docker 容器有以下两种运行态。

- ❑ 前台交互式：容器运行在前台，容器运行时直接连接到了程序中运行的程序上，当我们通过命令退出和关闭连接的程序时，就意味着容器停止了运行。在这种场景下，我们通常会通过附加的参数打开容器的伪终端和输入流，这样我们就可以和容器中的程序实现交互了。
- ❑ 后台守护式：容器运行在后台，运行的过程不会占用到当前输入指令的终端，也不会连接到容器内的应用程序上。因为我们无法连接到容器中的程序，所以处于这种运行态的容器必须通过指令来关闭。

在默认情况下，如果我们在使用 `docker run` 命令的时候没有显式指定容器的运行态，容器会以前台交互的形式运行，输入命令的终端会被挂起，其实这时我们已经连接到了容器中正在运行的应用程序上，当我们使用“`Ctrl + C`”组合键发出程序的终止信号后，程序会随之停止，容器也会随着程序的结束而停止。

虽然容器在前台运行，但我们只是连接到了容器中的应用程序，并没有形成与应用程序完整的交互，要与容器形成完整的交互，还需要使用 `-t` 和 `-i` 两个参数。在 `docker run` 命令中携带 `-t` 或 `--tty` 参数，可以让 Docker 为这个容器分配一个伪终端，为实现交互提供基础。而带入 `-i` 或 `--interactive` 则打开了交互模式，这时候输入流会一直保持，以便于我们向容器中输入命令或数据。

这里通过 Bash 进入 Debian 系统，并通过 `-i` 和 `-t` 准备好交互模式：

```
$ sudo docker run -i -t debian:jessie /bin/bash
root@d43c4a3361e0:/#
```

执行命令后，容器中的 Bash 就在等待输入指令了。通过检查系统版本，可以查看我们是否已经进入到了容器中：

```
root@d43c4a3361e0:/# uname -a
Linux d43c4a3361e0 4.4.15-moby #1 SMP Thu Jul 28 21:30:50 UTC 2016 x86_64 GNU/Linux
```

在容器中执行完操作之后，可以使用 `exit` 终止 Bash 程序，容器也会随之停止，我们就回到了宿主机中：

```
root@d43c4a3361e0:/# exit
exit
$
```

因为 Docker 在更多场合下是用于部署服务程序的，所以我们需要通过 `-d` 指令让容器运行在后台，这样不会占用和阻塞当前输入命令的终端，当我们退出终端后，容器仍会继续运行下去。

```
$ sudo docker run -d nginx:1.10
```

我们无法让容器在前后台同时运行，所以我们也就不能同时传入-d 和-i 参数，这两者是会产生冲突的。

小提示：在使用 Docker CLI 时，一些命令参数是可以进行组合的，比如-i 和-t 参数可以简写为-it，两者的效果是完全一样的。

### 3.1.2 容器的启动过程

在我们创建容器时，Docker 会先检查本地镜像库中是否有指定用于创建容器的镜像，如果没有在本地镜像库中找到镜像，Docker 会先从远程仓库中查找并下载到本地。接着，Docker 会创建容器的实例，将镜像以只读的方式挂载在为容器分配的文件系统上，并在只读镜像层的外侧，创建一个读写的层。之后，Docker 会配置容器的网络，连接到宿主主机中专用的网桥上，并为容器分配网络地址。

当我们启动已经创建好的容器时，所指定的应用程序会被运行，且容器会绑定到这个程序的启动进程上。当容器所绑定的进程结束时，容器也会随之停止。图 3-1 展示了容器启动和执行的过程。

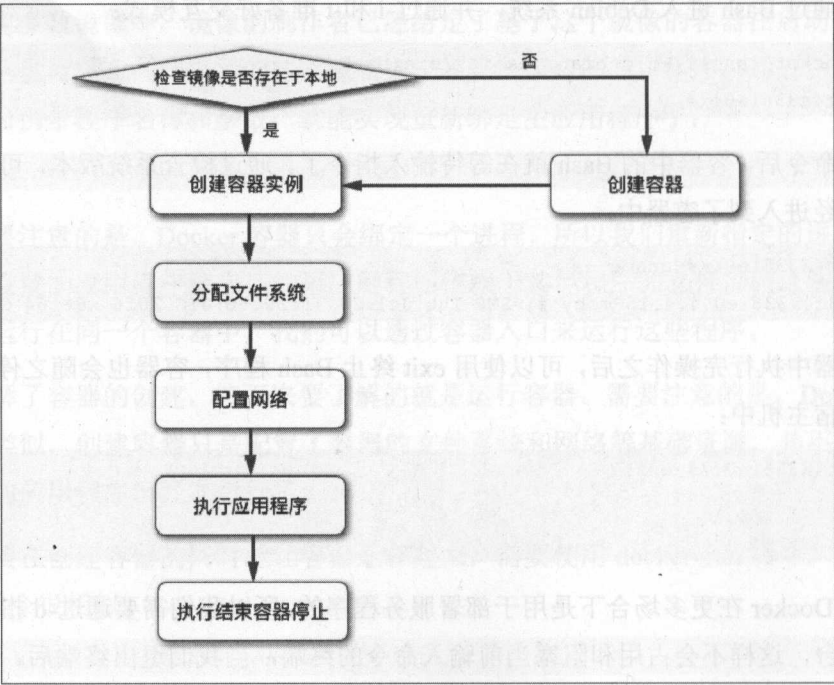


图 3-1 容器启动和执行的过程



### 3.1.3 列出容器

在 Linux 中，我们可以使用 `ps` 命令来查看系统中运行的进程；而在 Docker 中，我们也能通过 `docker ps` 命令来查看所有的容器。

```
$ sudo docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
PORTS		NAMES		
8a13693f2ba6	nginx:1.10	"nginx -g 'daemon off'"	2 weeks ago	Up 6 seconds
	443/tcp, 0.0.0.0:8080->80/tcp	nginx		
2a987e2d7153	php:7-fpm	"php-fpm"	2 weeks ago	Up 7 seconds
	9000/tcp	phpfpm		

在 `docker ps` 命令输出的字段中，可以看到以下信息。

- ❑ **CONTAINER ID:** 容器 ID。
- ❑ **IMAGE:** 容器所使用的镜像名称。
- ❑ **COMMAND:** 启动容器时应用程序的指令。
- ❑ **CREATED:** 容器的创建时间。
- ❑ **STATUS:** 容器运行的状态和更新状态的时间。Up 表示容器正在运行，如果容器处于暂停状态，会在表示状态的 Up 和状态的更新时间后，存在 Paused 字样。而 Exited 表示容器已经退出，在之后会有容器内关联进程退出的返回值。
- ❑ **PORTS:** 容器对外敞开的端口，我们会在后面专门介绍容器对外的端口和如何配置容器间的网络访问。
- ❑ **NAMES:** 容器的名称，是我们识别容器的另一种方式。

使用 `docker ps` 命令只会列出正在运行中的容器，如果要列出所有容器，可以通过携带 `-a` 或 `--all` 参数来实现。

```
$ sudo docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	PORTS	NAMES	
8a13693f2ba6	nginx:1.10	"nginx -g 'daemon off'"	2 weeks ago
Up 2 hours	443/tcp, 0.0.0.0:8080->80/tcp	nginx	
2a987e2d7153	php:7-fpm	"php-fpm"	2 weeks ago
Up 2 hours	9000/tcp	phpfpm	
96ab436b10bb	ubuntu	"/bin/bash"	2 weeks ago
Exited (0) 13 seconds ago		ubuntu	

由上可以看到，列出的容器中出现了已经停止的容器，在容器状态里，显示为

Exited(0) 13 seconds ago。其中，Exited(0)中的 0 表示容器结束时主进程的退出码，通常情况下为正常退出；13 seconds ago 表示容器上次更新状态的时间，也就是容器退出的时间。

当我们创建了较多容器时，使用 `docker ps` 命令会返回非常多的结果，在密集的结果中找到想要查看的容器并不是一件容易的事情，此时可以通过一些参数对 `docker ps` 的结果进行过滤。

使用 `-l` 或 `--latest` 参数可以列出最后创建的容器：

```
$ sudo docker ps -l
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
PORTS		NAMES		
8a13693f2ba6	nginx:1.10	"nginx -g 'daemon off'"	2 weeks ago	Up 2 hours
443/tcp, 0.0.0.0:8080->80/tcp		nginx		

使用 `-n <n>` 或 `--last <n>` 参数可以列出数个最近创建的容器：

```
$ sudo docker ps -n 2
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
PORTS		NAMES		
8a13693f2ba6	nginx:1.10	"nginx -g 'daemon off'"	2 weeks ago	Up 2 hours
443/tcp, 0.0.0.0:8080->80/tcp		nginx		
2a987e2d7153	php:7-fpm	"php-fpm"	2 weeks ago	Up 2 hours
9000/tcp		phpfpm		

使用 `-f "key=value"` 或者 `--filter "key=value"` 参数可以进行多种形式的过滤，例如过滤容器的名称：

```
$ sudo docker ps -f "name=phpfpm"
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
2a987e2d7153	php:7-fpm	"php-fpm"	2 weeks ago	Up 2 hours	9000/tcp	phpfpm

过滤的方式有很多种，都是在 `-f` 参数之后连接 `key=value` 的，进行多种过滤可以连用多个 `-f` 参数来实现，以下是可以被使用的过滤方式的键名（key）。

- ☐ `id`: 容器的 ID。
- ☐ `label`: 容器的标记，可以来源于创建容器的镜像，也可以在使用 `docker create` 或 `docker run` 创建容器时通过 `--label` 参数带入。
- ☐ `name`: 容器的名称。
- ☐ `exited`: 容器停止时主进程的返回码，进行 `exited` 过滤时需要同时使用 `-a` 或者 `--all` 确保查询到所有的容器。

- ❑ **status**: 容器的状态, 可以为 **created** (已创建)、**restarting** (重启中)、**running** (运行中)、**paused** (暂停中)、**exited** (已停止)、**dead** (已结束)。
- ❑ **ancestor**: 创建容器的镜像, 可以指定镜像名或镜像 ID。
- ❑ **before**: 给出一个容器名或容器 ID, 返回所有在给定容器之前创建的容器。
- ❑ **since**: 给出一个容器名或容器 ID, 返回所有在给定容器之后创建的容器。
- ❑ **isolation**: 隔离性, 可以为 **default** (默认)、**process** (进程隔离)、**hyperv** (虚拟机隔离), 这个参数只对 Windows 中的 Docker 有效。
- ❑ **volume**: 给出数据卷的名称或挂载点, 返回使用指定数据卷的容器。
- ❑ **network**: 给出网络 ID 或网络名称, 返回连接到指定网络的容器。

### 3.1.4 容器的命名

在 `docker ps` 列出的容器信息中, 可以看到 **NAMES** 字段, 即容器的名称。容器名称与镜像名称类似, 也是除容器 ID 以外另一种识别容器的方式, 在同一个 `docker daemon` 下, 容器名称是唯一的, 在进行大部分操作时, 可以使用容器名称来替代容器 ID。

给容器命名最大的好处就是名称是一个可以表意的字符串, 相较于随机生成的容器 ID, 其可读性更高。可以使用表示容器用途的名字为容器命名, 方便我们进行记忆。

在默认情况下, 如果我们没有为创建的容器命名, Docker 会用单词组合的形式为容器取一个随机的名字。

```
$ sudo docker run -d nginx:stable
1014a82f7c2fld3a9371bd4abcc62f3992786f4e1aa78f469d2c109ad2a25c2b
$ sudo docker ps -l
```

CONTAINER ID	IMAGE	COMMAND	CREATED
1014a82f7c2f	nginx:stable	"nginx -g 'daemon off'"	4 seconds ago
Up 3 seconds	80/tcp, 443/tcp	modest_jang	

这样随机生成的名字虽然有一定的含义, 但还不能完全达到我们的目的。在创建容器时给出 `--name <name>` 参数, 可以自定义被创建容器的名称。

```
$ sudo docker run -d --name web nginx:stable
efcd69e060ee3107853bf58052b7665299277fbecbd13c2370c7fb2350700417
$ sudo docker ps -l
```

CONTAINER ID	IMAGE	COMMAND	CREATED
efcd69e060ee	nginx:stable	"nginx -g 'daemon off'"	3 seconds ago
Up 1 seconds	80/tcp, 443/tcp	web	



给容器命名还可以为我们之后建立容器间的互相依赖关系提供方便。例如，我们需要建立 Nginx + PHP + MySQL 的服务器架构，可以把三个容器分别命名为 nginx、php、mysql，这样在处理依赖时，我们可以很方便地指定 Nginx 容器连接到 PHP 容器、PHP 容器连接到 MySQL 容器。在进行容器连接和数据共享时，清晰的依赖关系是达到目的的关键。

### 3.1.5 启动和停止

我们可以通过 `docker run` 命令来创建并启动容器，但我们还需要启动通过 `docker create` 命令创建的容器，以及启动已经停止的容器，这时就需要使用 `docker start` 命令了。

```
$ sudo docker start web
web
```

使用 `docker start` 命令时，可以传入我们为容器设置的名字，也可以传入容器的 ID，当我们传入容器 ID 时，可以只传入容器 ID 的前一部分，只要能够判断容器在 Docker 中的唯一性即可。

```
$ sudo docker start 8a136
8a136
```

如果容器启动顺利，我们可以在终端中看到 `docker start` 命令返回的结果，即我们使用的容器名称或容器 ID。

在默认情况下，容器会以后台守护的方式启动运行，当然我们也可以切换到交互模式。在启动容器时使用 `-i` 或 `--interactive` 参数，可以将终端连接到容器中主应用程序的标准输入上，而携带 `-a` 或 `--attach` 参数，则可以将容器中程序的标准输入及错误输出的结果，显示到终端中。

如果容器组成了一套服务体系，我们可以使用一条 `docker start` 命令来启动它们，只要逐个传入容器名称或容器 ID 到 `docker start` 命令中即可。

```
$ sudo docker start nginx phpfpn mysql
nginx
phpfpn
mysql
```

使用 `docker stop` 命令可以停止正在运行中的容器，当我们发出 `docker stop` 命令时，Docker 会向容器所绑定的主要进程发送 `SIGTERM` 信号，告知程序自行停止。如果程序在限定的时间内没有自动停止，则 Docker 会发送 `SIGKILL` 信号强制结束进程。如我

们之前提到的，容器的生命周期是与主进程绑定的，所以在主进程结束后，容器也就停止了。

```
$ sudo docker stop web
web
```

Docker 提供给程序自行退出的默认时间是 10 秒，我们也可以通过传入 `-t` 或 `--time` 参数来更改这个时间。

```
$ sudo docker stop -t 3 web
web
```

另外，Docker 还提供了 `docker kill` 命令来直接停止容器，`docker kill` 命令会直接发送 SIGKILL 信号强制结束容器的进程。

```
$ sudo docker kill web
web
```

被 `docker kill` 命令传递到容器中的信号是可以自定义的，也可以通过传入 `-s` 或 `--signal` 参数定义被使用在进程上的信号。

```
$ sudo docker kill -s INT web
web
```

与 `docker start` 命令的作用一样，使用 `docker stop` 命令也可以同时传入多个容器：

```
$ sudo docker stop nginx phpfpn mysql
nginx
phpfpn
mysql
```

### 3.1.6 暂停和恢复

除了启动和停止容器，我们还可以让容器暂停或恢复，使用 `docker pause` 和 `docker unpause` 命令可以分别暂停和恢复容器的运行。

暂停容器与停止容器最大的区别在于，停止容器时容器中运行的应用程序也会一同关闭，而暂停容器时，容器中运行的应用程序只会被暂停。我们可以简单地把容器的停止与暂停理解为电脑的关机与休眠。

暂停容器和恢复容器的方法很简单：

```
$ sudo docker ps -l
```

CONTAINER ID	IMAGE	COMMAND	CREATED
--------------	-------	---------	---------

```
STATUS          PORTS          NAMES
7c13794de69c    nginx:stable   "nginx -g 'daemon off'" 35 seconds ago
Up 2 seconds    80/tcp, 443/tcp web
$ sudo docker pause web
web
$ sudo docker ps -l
CONTAINER ID    IMAGE          COMMAND          CREATED
STATUS          PORTS          NAMES
7c13794de69c    nginx:stable   "nginx -g 'daemon off'" 39 seconds ago
Up 6 seconds (Paused) 80/tcp, 443/tcp web
$ sudo docker unpause web
web
$ sudo docker ps -l
CONTAINER ID    IMAGE          COMMAND          CREATED
STATUS          PORTS          NAMES
7c13794de69c    nginx:stable   "nginx -g 'daemon off'" 42 seconds ago
Up 9 seconds    80/tcp, 443/tcp web
```

docker pause 和 docker unpause 命令也可以同时作用于多个容器:

```
$ sudo docker pause nginx phpfpn mysql
nginx
phpfpn
mysql
$ sudo docker unpause nginx phpfpn mysql
nginx
phpfpn
mysql
```

Docker 容器暂停的原理是使用 Linux 的 cgroup freezer 将容器中的所有进程都挂起。与传统的挂起进程的方式不同,使用 cgroup freezer 挂起进程对进程来说是透明的,程序不会感知到自己已经被暂停,也不会感知到自己被恢复。由于没有传统挂起进程时传递给进程的 SIGSTOP 信号,所以进程不知道自己将会被挂起,也就没有机会对正在进行的工作进行保存。

### 3.1.7 重启容器

Docker 还提供了 docker restart 命令用于重启容器。

重启容器的过程就是停止容器和再次启动容器的过程,相当于 docker stop 和 docker start 命令的组合,使用 docker restart 命令的方法也非常简单:



```
$ sudo docker restart web
web
```

### 3.1.8 删除容器

因为 Docker 特有的镜像机制，在创建容器的过程中没有大型的 IO 操作，所以创建容器的过程是秒级的。因此，我们更倾向于随用随删，在停止程序所在的容器时，将容器一并删除。

使用 `docker rm` 命令可以删除容器：

```
$ sudo docker rm web
web
```

直接使用 `docker rm` 命令时，需要保证被删除的容器处于停止状态，正在运行中的容器是不会被删除的。如果需要删除正在运行的容器，需要携带 `-f` 或 `--force` 参数：

```
$ sudo docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED
STATUS        PORTS          NAMES
7c13794de69c   nginx:stable   "nginx -g 'daemon off'" About an hour ago
Up 6 seconds   80/tcp, 443/tcp web
$ sudo docker rm web
Error response from daemon: You cannot remove a running container 7c13794de69cf816dade298f3fe3db1ca7da580e1ea6114fb25c1012180ba36a. Stop the container before attempting removal or use -f
$ sudo docker rm -f web
web
```

当我们强制删除容器时，Docker 会先向容器中的主进程发送 `SIGKILL` 信号，并强制停止容器，之后再对容器进行删除。

## 3.2 连接到容器

在很多场景下，我们需要了解容器的运行情况，甚至需要进入到容器中进行一些操作，此时就需要连接到容器。连接容器可以让我们获得并监视容器实时发生的事件，还可以通过输入/输出流，对容器内的程序下达指令，实现特定需求。

### 3.2.1 查看进程信息

容器运行起来以后，特别是以后台守护态运行起来以后，一个迫切的需求是通过查看容器中的信息，来了解容器运行的情况。Docker 为我们提供了很多查看容器中信息的方法，查看容器中运行的进程就是其中之一。

查看进程信息是监视程序运行情况最基本的一种方法，我们在各种操作系统中都会经常使用到。在 Linux 系统中，我们最常使用的就是 `ps` 和 `top` 两个命令，`ps` 和 `top` 命令都可以显示出当前系统运行的进程信息。同样的，Docker 也为我们提供了一个查看容器的命令——`docker top` 命令。

一看到这个名字，相信很多人会认为这个命令的用法与 Linux 系统中 `top` 命令的用法应该非常相似，然而恰恰相反，`docker top` 命令的结果更像在容器内执行 `ps` 命令的结果。

`docker top` 命令的使用方法很简单，只需要将要查看的容器名或容器 ID 传给 `docker top` 命令就可以了：

```
$ sudo docker top web
```

ID	PID	PPID	C	STIME	TTY	TIME	CMD
root	598	1544	0	3:03	?	00:00:00	nginx: master process nginx
nginx	1674	1598	0	3:03	?	00:00:01	nginx: worker process

为什么说 `docker top` 命令更像 `ps` 命令呢？首先，`docker top` 命令执行的结果是展示出容器内运行进程的列表，并不像 `top` 命令那样进入实时监控的状态。其次，`docker top` 命令可以接收 `ps` 命令的参数，我们可以在容器名或容器 ID 之后加上 `ps` 命令所支持的参数，这样就能像在容器中使用 `ps` 命令一样获得想要的结果了。

这里我们使用 `ps` 命令的 `-x` 参数来显示没有 TTY（虚拟控制台）的进程：

```
$ sudo docker top web -x
```

PID	TTY	STAT	TIME	COMMAND
1598	?	Ss	0:00	nginx: master process nginx

### 3.2.2 查看容器信息

对于已经创建的容器，容器的环境变量、运行命令、网络配置等也是我们了解容器非常重要的信息。在 Docker 中，我们可以使用 `docker inspect` 命令来查看容器的信息。

```

$ sudo docker inspect web
[
  {
    "Id": "d68d9aaf5d05036319428a3f7bc31c0549afb41a447584e1607e547b06ea5c36",
    "Created": "2016-08-06T12:59:20.186828303Z",
    "Path": "nginx",
    "Args": [
      "-g",
      "daemon off;"
    ],
    "State": {
      "Status": "running",
      "Running": true,
      "Paused": false,
      "Restarting": false,
      "OOMKilled": false,
      "Dead": false,
      "Pid": 1791,
      "ExitCode": 0,
      "Error": "",
      "StartedAt": "2016-08-07T01:49:48.598841751Z",
      "FinishedAt": "2016-08-06T14:30:44.063176377Z"
    },
    "Image": "sha256:82e97a2b0390a20107ab1310dea17f539ff6034438099384998fd91fc540b128",
    "ResolvConfPath": "/var/lib/docker/containers/d68d9aaf5d05036319428a3f7bc31c0549afb41a447584e1607e547b06ea5c36/resolv.conf",
    "HostnamePath": "/var/lib/docker/containers/d68d9aaf5d05036319428a3f7bc31c0549afb41a447584e1607e547b06ea5c36/hostname",
    "HostsPath": "/var/lib/docker/containers/d68d9aaf5d05036319428a3f7bc31c0549afb41a447584e1607e547b06ea5c36/hosts",
    "LogPath": "/var/lib/docker/containers/d68d9aaf5d05036319428a3f7bc31c0549afb41a447584e1607e547b06ea5c36/d68d9aaf5d05036319428a3f7bc31c0549afb41a447584e1607e547b06ea5c36-json.log",
    "Name": "/web",
    "RestartCount": 0,
    "Driver": "aufs",
    "MountLabel": "",
    "ProcessLabel": "",
    "AppArmorProfile": "",
    "ExecIDs": null,
    "HostConfig": {

```



```
"Binds": null,
"ContainerIDFile": "",
"LogConfig": {
  "Type": "json-file",
  "Config": {}
},
"NetworkMode": "default",
"PortBindings": {},
"RestartPolicy": {
  "Name": "no",
  "MaximumRetryCount": 0
},
"AutoRemove": false,
"VolumeDriver": "",
"VolumesFrom": null,
"CapAdd": null,
"CapDrop": null,
"Dns": [],
"DnsOptions": [],
"DnsSearch": [],
"ExtraHosts": null,
"GroupAdd": null,
"IpcMode": "",
"Cgroup": "",
"Links": null,
"OomScoreAdj": 0,
"PidMode": "",
"Privileged": false,
"PublishAllPorts": false,
"ReadonlyRootfs": false,
"SecurityOpt": null,
"UTSMode": "",
"UsernsMode": "",
"ShmSize": 67108864,
"Runtime": "runc",
"ConsoleSize": [
  0,
  0
],
"Isolation": "",
"CpuShares": 0,
"Memory": 0,
```

```

    "CgroupParent": "",
    "BlkioWeight": 0,
    "BlkioWeightDevice": null,
    "BlkioDeviceReadBps": null,
    "BlkioDeviceWriteBps": null,
    "BlkioDeviceReadIOps": null,
    "BlkioDeviceWriteIOps": null,
    "CpuPeriod": 0,
    "CpuQuota": 0,
    "CpusetCpus": "",
    "CpusetMems": "",
    "Devices": [],
    "DiskQuota": 0,
    "KernelMemory": 0,
    "MemoryReservation": 0,
    "MemorySwap": 0,
    "MemorySwappiness": -1,
    "OomKillDisable": false,
    "PidsLimit": 0,
    "Ulimits": null,
    "CpuCount": 0,
    "CpuPercent": 0,
    "IOMaximumIOps": 0,
    "IOMaximumBandwidth": 0
  },
  "GraphDriver": {
    "Name": "aufs",
    "Data": null
  },
  "Mounts": [],
  "Config": {
    "Hostname": "d68d9aaf5d05",
    "Domainname": "",
    "User": "",
    "AttachStdin": false,
    "AttachStdout": false,
    "AttachStderr": false,
    "ExposedPorts": {
      "443/tcp": {},
      "80/tcp": {}
    },
    "Tty": false,

```



```
"OpenStdin": false,
"StdinOnce": false,
"Env": [
  "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
  "NGINX_VERSION=1.10.1-1~jessie"
],
"Cmd": [
  "nginx",
  "-g",
  "daemon off;"
],
"Image": "nginx:stable",
"Volumes": null,
"WorkingDir": "",
"Entrypoint": null,
"OnBuild": null,
"Labels": {}
},
"NetworkSettings": {
  "Bridge": "",
  "SandboxID": "01b85066b5a4650221ea3f28a0b481d4e5fffb9a4bc0f1ff341e9de5205
cefc2c",
  "HairpinMode": false,
  "LinkLocalIPv6Address": "",
  "LinkLocalIPv6PrefixLen": 0,
  "Ports": {
    "443/tcp": null,
    "80/tcp": null
  },
  "SandboxKey": "/var/run/docker/netns/01b85066b5a4",
  "SecondaryIPAddresses": null,
  "SecondaryIPv6Addresses": null,
  "EndpointID": "62c29682b34a7e9c39c13543a82fdaba3435a78dd9495513b3a8bb70b
64c939a",
  "Gateway": "172.17.0.1",
  "GlobalIPv6Address": "",
  "GlobalIPv6PrefixLen": 0,
  "IPAddress": "172.17.0.2",
  "IPPrefixLen": 16,
  "IPv6Gateway": "",
  "MacAddress": "02:42:ac:11:00:02",
  "Networks": {
```



```

        "bridge": {
            "IPAMConfig": null,
            "Links": null,
            "Aliases": null,
            "NetworkID": "9a0015124b4401138ea8a2d304216617e6f3d18d49ad23d663
aa3107a4ec084f",
            "EndpointID": "62c29682b34a7e9c39c13543a82fdaba3435a78dd9495513b3
a8bb70b64c939a",
            "Gateway": "172.17.0.1",
            "IPAddress": "172.17.0.2",
            "IPPrefixLen": 16,
            "IPv6Gateway": "",
            "GlobalIPv6Address": "",
            "GlobalIPv6PrefixLen": 0,
            "MacAddress": "02:42:ac:11:00:02"
        }
    }
}
]

```

我们可以看到 `docker inspect` 命令返回了容器的很多信息，主要包含以下信息。

- ☐ Id: 容器 ID。
- ☐ Created: 容器创建时间。
- ☐ Path: 容器启动命令。
- ☐ Args: 启动命令附带的参数。
- ☐ State: 容器的状态。
- ☐ Image: 容器所基于的镜像，展示的是镜像的校验码。
- ☐ ResolvConfPath: 容器的配置文件路径。
- ☐ HostnamePath: 容器的 `hostname` 文件的存储位置。
- ☐ HostsPath: 容器的 `hosts` 文件的存储位置。
- ☐ LogPath: 容器日志记录文件。
- ☐ Name: 容器的名称。
- ☐ RestartCount: 容器的重启次数。
- ☐ Driver: 文件系统驱动。
- ☐ MountLabel: 挂载标签。
- ☐ ProcessLabel: 进程标签。

- ☐ AppArmorProfile: 安全配置。
- ☐ ExecIDs: 执行 ID。
- ☐ HostConfig: 主机配置。
- ☐ GraphDriver: 图形驱动。
- ☐ Mounts: 挂载数据卷。
- ☐ Config: 容器配置。
- ☐ NetworkSettings: 网络配置。

`docker inspect` 命令也可以接受容器 ID 及容器 ID 前一部分作为参数，但需要注意的是，`docker inspect` 命令能够查看容器、镜像及任务的信息，所以在传入容器名称、容器 ID 前一部分时，需要保证这些信息不会与镜像、任务出现相同的不可区分的状况。

另外，我们也可以使用 `-f` 或 `--format` 参数来过滤想要查看的信息。例如查看容器的运行状态：

```
$ sudo docker inspect -f "{{ .State.Running }}" web
true
```

通过 `docker inspect` 命令，我们可以查阅到容器各方面的信息，这对于了解容器的运行状态、判断容器出现的问题、对容器进行配置都有很大的帮助。

### 3.2.3 容器日志

对于前台交互型容器，我们可以在输入命令的终端直接看到运行在容器中的应用程序输出的结果，但是以后台守护形式启动的容器，程序输出的结果不会直接显示在终端中。这时，我们可以通过 `docker logs` 命令查看应用程序输出的内容。

为了方便、清晰地了解 `docker logs` 命令的特性，我们先构造一个不断输出结果的容器：

```
$ sudo docker run -d --name logs_demo ubuntu /bin/bash -c 'for((i=0;1;i++));do echo "time $i";sleep 1;done;'
```

在这个容器中，我们通过 `Bash` 执行脚本，每秒钟输出递增的秒数到程序的输出流中，而容器是以守护态启动的，所以我们在输入命令的终端上不能直接看到程序输出的信息。这时我们通过 `docker logs` 命令来查询容器输出的内容。

```
$ sudo docker logs logs_demo
time 0
time 1
```

```
time 2
time 3
time 4
time 5
```

由此可以看到，容器内应用程序输出的结果被展示了出来。

在默认情况下，`docker logs` 会展示出自程序启动到执行 `docker logs` 时所有的输出，然后停止。如果我们要持续显示程序的输出，可以使用 `-f` 或 `--follow` 参数，这样就能持续看到程序输出的内容了。

当程序输出的内容较多时，我们可以通过 `--tail` 和 `--since` 限制输出的内容。使用 `--tail` 参数可以选择输出内容的最后几行进行返回，使用 `--since` 参数则可以限定只返回指定时间戳之后输出的内容。两个参数的使用方法如下。

```
$ sudo docker logs --tail 5 logs_demo
time 703
time 704
time 705
time 706
time 707
$ sudo docker logs --since 2016-08-07T05:56:38.192382641Z logs_demo
time 726
time 727
time 728
time 729
time 730
```

当我们连用 `-f` 和 `--tail` 参数时，可以达到不输出之前内容的效果。比如我们想从当前时间开始查看输出内容：

```
$ sudo docker logs -f --tail 0 logs_demo
time 824
time 825
...
```

容器输出的结果不会再包含输出前的结果，只是把从这个时间开始输出的内容打印到屏幕上。

### 3.2.4 衔接到容器

当容器以守护形态运行起来以后，虽然我们能够查阅容器的进程信息和应用程序输



出的内容，但这些操作其实还只是在容器外进程的操作，还没有真正进入到容器中去。所谓进入到容器中，就是我们直接对容器中的主程序进行操作，或者对容器中的系统及其他程序进行操作。Docker 为我们提供了进入容器的方法，`docker attach` 命令就是其中之一。

`docker attach` 命令可以让我们进入到容器中，并将当前输出命令的终端连接到容器中主进程的标准输入与标准输出中。使用了 `docker attach` 命令之后，就相当于将容器切换到了前台交互模式。

```
$ sudo docker attach web
```

需要注意的是，`docker attach` 命令绑定终端到容器的主程序上，所以我们可以持续收到进程输出的内容。与此同时，当我们使用“Ctrl + C”等组合键发出停止信号时，收到命令的其实是容器中的应用程序。

```
$ sudo docker start web
web
$ docker attach web
^C
$ sudo docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
--------------	-------	---------	---------	--------	-------	-------

也就是说，我们在使用 `docker attach` 命令连接到容器后，所执行的操作就会和直接使用交互形式启动容器一样，所有的操作都是针对容器中的应用程序来进行的，包括程序的停止。

### 3.2.5 在容器中执行命令

使用 `docker attach` 命令可以衔接到容器中运行的主进程上，并且衔接后就依附到了主进程之中，当我们退出程序时，也会导致容器一起停止。那么有没有其他的办法来实现在容器中进行操作呢？Docker 提供了 `docker exec` 命令来达成我们的需求。

`docker exec` 命令可以让我们在容器中执行一条全新的指令，并开启新的进程。使用 `docker exec` 命令的方法很简单，给出容器名称或容器 ID 并提供需要执行的命令即可。

```
$ sudo docker exec web ps
```

PID	TTY	TIME	CMD
1	?	00:00:00	nginx
6	?	00:00:00	ps

在默认情况下, `docker exec` 会将输入命令的终端连接到执行程序的输出流中, 所以如果程序持续执行下去, 我们会在终端中看到程序输出的结果不断显示出来。

```
$ sudo docker exec web tail /var/log/nginx/access.log
```

如果我们不希望终端连接到容器输出, 而是静默地跟随容器继续在后台执行, 只要添加 `-d` 或 `--detach` 参数即可。

```
$ sudo docker exec -d web tail /var/log/nginx/access.log
```

当然, 我们还可以通过 `-i` 或 `--interactive` 打开输入流进入交互模式, 而带入 `-t` 或 `--tty` 则可以分配伪终端给程序使用。通常可以直接以交互方式执行 `Bash` 或其他 `Shell` 程序, 这样就能直接进入到容器中, 就像在容器中的系统里进行交互一样。

```
$ sudo docker exec -it web /bin/bash
root@d68d9aaf5d05:/# ps
PID TTY          TIME          CMD
  7  ?            00:00:00      bash
 12  ?            00:00:00      ps
root@d68d9aaf5d05:/# exit
```

使用 `docker exec` 命令可以在容器中打开其他程序, 为我们更好地查看容器运行信息、对容器进行维护和管理等操作提供指导。

## 3.3 容器的保存与迁移

在某些场景下, 需要把容器中的应用程序及其生成的数据转移到其他主机中。有时, 我们也期望容器数据可以持久保存下来, 以进行备份和恢复。这些需求我们可以通过容器的保存来实现。

### 3.3.1 提交容器更改

假设我们希望拥有一个属于自己的基于 `Ubuntu` 系统的 `Nginx` 服务器镜像, 那么需要如何制作这个镜像呢?

容器中所有的更改都不会作用于基础镜像中, 而在可读写层上的修改又存在于沙盒环境下, 当容器停止后, 这些修改也会丢失, 那么怎样才可以保存这些修改呢?

Docker 为我们提供了将容器中可读性层的修改保存为新的镜像层的方法，通过 `docker commit` 命令，我们可以很轻松地把容器中的修改保存并生成一个新的镜像。

先启动一个 CentOS 容器，然后使用 `yum` 安装 `crontabs` 定时任务程序，之后结束交互，停止容器的运行。

```
$ sudo docker run -it centos:7 /bin/bash
[root@a278b7c6f184 /]# yum install -y crontabs
Loaded plugins: fastestmirror, ovl
base                                     | 3.6 kB    00:00
extras                                 | 3.4 kB    00:00
updates                                | 3.4 kB    00:00
(1/4): base/7/x86_64/group_gz          | 155 kB    00:00
(2/4): extras/7/x86_64/primary_db       | 160 kB    00:02
(3/4): updates/7/x86_64/primary_db      | 6.4 MB    00:33
(4/4): base/7/x86_64/primary_db         | 5.3 MB    00:37
Determining fastest mirrors
* base: mirrors.163.com
* extras: centos.ustc.edu.cn
* updates: mirrors.163.com
Resolving Dependencies
--> Running transaction check
---> Package crontabs.noarch 0:1.11-6.20121102git.el7 will be installed
--> Processing Dependency: /etc/cron.d for package: crontabs-1.11-6.20121102git.el7.noarch
--> Running transaction check
---> Package cronie.x86_64 0:1.4.11-14.el7_2.1 will be installed
--> Processing Dependency: dailyjobs for package: cronie-1.4.11-14.el7_2.1.x86_64
--> Running transaction check
---> Package cronie-anacron.x86_64 0:1.4.11-14.el7_2.1 will be installed
--> Finished Dependency Resolution

Dependencies Resolved

=====
Package            Arch      Version                               Repository    Size
=====
Installing:
crontabs           noarch    1.11-6.20121102git.el7               base          13 k
Installing for dependencies:
cronie             x86_64    1.4.11-14.el7_2.1                   updates       90 k
cronie-anacron     x86_64    1.4.11-14.el7_2.1                   updates       35 k
```



## Transaction Summary

```
=====
Install 1 Package (+2 Dependent packages)
```

```
Total download size: 138 k
```

```
Installed size: 259 k
```

```
Downloading packages:
```

```
warning: /var/cache/yum/x86_64/7/updates/packages/cronie-anacron-1.4.11-14.el7_2.1.x86_64.rpm: Header V3 RSA/SHA256 Signature, key ID f4a80eb5: NOKEY
```

```
Public key for cronie-anacron-1.4.11-14.el7_2.1.x86_64.rpm is not installed
```

```
(1/3): cronie-anacron-1.4.11-14.el7_2.1.x86_64.rpm | 35 kB 00:00
```

```
(2/3): cronie-1.4.11-14.el7_2.1.x86_64.rpm | 90 kB 00:02
```

```
Public key for crontabs-1.11-6.20121102git.el7.noarch.rpm is not installed
```

```
(3/3): crontabs-1.11-6.20121102git.el7.noarch.rpm | 13 kB 00:02
```

```
-----
Total 51 kB/s | 138 kB 00:02
```

```
Retrieving key from file:///etc/pki/rpm-gpg/RPM-GPG-KEY-CentOS-7
```

```
Importing GPG key 0xF4A80EB5:
```

```
Userid : "CentOS-7 Key (CentOS 7 Official Signing Key) <security@centos.org>"
```

```
Fingerprint: 6341 ab27 53d7 8a78 a7c2 7bb1 24c6 a8a7 f4a8 0eb5
```

```
Package : centos-release-7-2.1511.el7.centos.2.10.x86_64 (@CentOS)
```

```
From : /etc/pki/rpm-gpg/RPM-GPG-KEY-CentOS-7
```

```
Running transaction check
```

```
Running transaction test
```

```
Transaction test succeeded
```

```
Running transaction
```

```
Installing : crontabs-1.11-6.20121102git.el7.noarch 1/3
```

```
Installing : cronie-1.4.11-14.el7_2.1.x86_64 2/3
```

```
Installing : cronie-anacron-1.4.11-14.el7_2.1.x86_64 3/3
```

```
Verifying : cronie-anacron-1.4.11-14.el7_2.1.x86_64 1/3
```

```
Verifying : crontabs-1.11-6.20121102git.el7.noarch 2/3
```

```
Verifying : cronie-1.4.11-14.el7_2.1.x86_64 3/3
```

```
Installed:
```

```
crontabs.noarch 0:1.11-6.20121102git.el7
```

```
Dependency Installed:
```

```
cronie.x86_64 0:1.4.11-14.el7_2.1 cronie-anacron.x86_64 0:1.4.11-14.el7_2.1
```

```
Complete!
```

```
[root@a278b7c6f184 /]# exit
```

```
exit
```

我们找出刚才操作的容器 ID:

```
$ sudo docker ps -l -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
b8cb65773346	centos:7	"/bin/bash"	About a minute ago	Exited (0) 2 seconds ago

```
clever_noether
```

之后我们就能使用 `docker commit` 命令来生成新的镜像了。

```
$ sudo docker commit b8cb65773346
sha256:0a09b20ad46acb34ed6ba980c074ee64c90269e358e6ef4f0d7d0ab5a2ff8dea
```

`docker commit` 命令会返回刚刚创建的镜像的 ID，我们也可以在本地图像库中看到刚刚创建的镜像。

```
$ sudo docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
<none>	<none>	0a09b20ad46a	15 seconds ago	289.2 MB

```
...
```

还可以使用 `-m` 参数为镜像层撰写一条提交信息，也可以为镜像设置镜像名。

```
$ sudo docker commit -m "Crontabs" b8cb65773346 ymd/crontab:latest
sha256:19ae65a7796f288a4828cf97754cef0996f88c797a5419eb0e57a796842ec3c5
```

```
$ sudo docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
ymd/crontab	latest	19ae65a7796f	14 seconds ago	289.2 MB

```
...
```

### 3.3.2 容器的导入/导出

包括 Docker 在内的虚拟化技术，进行快速迁移是重要的目标之一，前面的章节中谈到了镜像的迁移，在某些场合下也需要对容器进行迁移。我们了解了将容器提交成镜像的方法，结合导入/导出容器的命令，可以实现将容器转换成镜像，再通过镜像运行成容器完成迁移。不过这种方式过于烦琐，还要浪费时间去处理生成的这些无用镜像。好在 Docker 也为容器提供了导入/导出的方法，通过容器的导入/导出命令，让我们可以轻松地完成对容器的迁移。

不论容器是否处于运行的状态，都可以使用 `docker export` 命令将容器保存到一个压缩文件中。

```
$ sudo docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
d68d9aaf5d05	nginx:stable	"nginx -g 'daemon off'"	3 days ago	Up 3 seconds
80/tcp, 443/tcp	web			

```
$ sudo docker export d68d9aaf5d05 >web.tar
```

和导出镜像时使用的 `docker save` 命令一样, `docker export` 命令默认将导出的容器数据放置到输出流中, 所以需要使用 `>web.tar` 将输出流接收到文件中去。当然, 也可以像 `docker save` 命令一样使用 `-o` 或 `--output` 参数指定输出文件, 让 `docker export` 命令不再输出数据。

```
$ sudo docker export -o web.tar d68d9aaf5d05
```

我们可以将保存容器数据的压缩文件通过网络或其他方式传输到目标机器上, 并进行容器数据的恢复, 不用担心容器所依赖的镜像问题, 所有容器中的文件数据都已经包含在了导出的数据之中。

想要导入容器的数据, 可以使用 `docker import` 命令。需要注意的是, 导入的容器不是直接出现在主机的容器列表中, 而是出现在本地镜像库里, 因为当容器被导出时, 容器运行时所在的沙盒环境就已经静默地被持久化成镜像层了。

```
$ sudo docker import web.tar
```

```
sha256:17d4086d283048d9ac3cd16337dc774817587b8394a7080017d7c86463617c79
```

```
$ sudo docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
<none>	<none>	17d4086d2830	About a minute ago	181.3 MB

虽然容器的导入和镜像的导入都是导入到主机的本地镜像库中, 但两者还是有区别的, 因为容器的导出文件与镜像的导出文件在结构上是有差异的。容器的导出文件着重记录容器导出时容器内的状况, 包括容器中所有的文件; 而镜像的导出文件, 除了记录所有镜像层的文件, 还保持着镜像原有的元数据信息。所以, 在使用空间上镜像的导出文件一般会比容器的导出文件大一些。

`docker import` 命令还支持从网络地址导入, 只要将文件路径换成网络地址即可。如果需要对新的镜像设置名称, 也可以在文件或网络路径后加上想要设置的镜像名。

```
$ sudo docker import web.tar ymd/web:1.0
```

```
sha256:470b41a5fbf2386f05816d391f9032c606089735a08d30516046b5b8f48e3276
```

```
$ sudo docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
ymd/web	1.0	470b41a5fbf2	4 seconds ago	181.3 MB



## 3.4 本章小结

容器的 Docker 是最核心的模块，也几乎是所有其他模块的基础，掌握好容器的知识和容器的操作，有利于我们继续探索 Docker 的其他模块。

在本章中，我们通过讲解和示例，了解了 Docker 容器的生命周期，掌握了 Docker 容器的使用方法。绝大部分 Docker 操作都是围绕容器展开的，所以熟练地使用容器是 Docker 的基础，当我们熟悉了容器的操作之后，后面章节的学习就会变得非常容易了。

# 第 4 章

## 数据卷与网络

当我们的程序在容器中工作时，特别是需要与其他容器中的程序或容器外部程序进行沟通交流时，就需要进行数据的交换。作为最常使用的两种沟通数据的方式，网络通信与文件读写无疑是需要提供给程序的基础支持。在这两方面，Docker 也为我们提供了强有力的支持。

在本章中，我们将分别介绍 Docker 为文件持久化和共享所提供的数据卷，以及为网络访问和容器间建立网络连接所提供的支持。

### 4.1 数据卷

文件是数据持久化最常用的保存方法，由于容器内文件系统的隔离，以及其本身是以沙盒形式运行等性质，使其对保存需要持久化的数据并不理想和稳定。对此，Docker 使用了数据卷这种专门定制的形式，来提供一套便于持久化保存数据的体系。本节我们就专门介绍数据卷——用于存储文件数据的特殊模块。

#### 4.1.1 关于数据卷

读取和写入文件，是程序运行中获取外界信息和向外界输出信息最简便的方式之一。

从文件中读取配置或向文件里写入日志，几乎所有的程序都或多或少地使用到了文件的读写等操作，更何况大部分的程序都是以二进制或文本的形式保存在文件中。容器也有完整的文件系统，而 Docker 也为运行在容器中的应用程序，提供了全面的支持，应用程序虽然在容器之中运行，但也具有对容器内文件的完全控制和操作能力。我们在容器中使用和文件相关的操作，就和在真实机器上使用这些操作的方法和所得到的结果是完全一致的，所以对文件的操作并不是本书想要说明的重点，介绍 Docker 文件体系的重点在于文件持久化的问题。

容器内的文件环境，是由联合文件系统提供的一个临时层，虽然能够让程序随意操作其中的文件，但所有的读写都是在沙盒环境中进行的。当容器停止运行并被删除时，这个临时记录着文件修改的层就会被一同丢弃。即使我们能够通过提交镜像的方式保存容器中文件的修改，但利用这种方式实现文件的持久化不但操作过于烦琐，也不便于进行自动化管理，仍然达不到我们想要的效果。更何况通过镜像的方式保存文件，只解决了文件的输出问题，并没有解决文件输入的问题。

为了达到从外界获取文件以及持久化存储文件的目的，Docker 提出了数据卷（Data Volume）的概念。简而言之，数据卷就是一个挂载在容器内文件系统中的文件或目录。在容器中，数据卷和其他的文件或目录看起来别无二致，但是因为数据卷是从外界挂载在容器中的，所以它可以脱离容器的生命周期而独立存在。正是由于数据卷的生命周期并不等同于容器的生命周期，在容器退出乃至删除之后，数据卷仍不会受到影响，会依然存在于 Docker 之中。所以，如果通过数据卷保存文件，那么这些文件就不会因为容器的终结而消失。

### 4.1.2 数据卷的特点

数据卷作为容器中一个特殊的文件或目录，与容器中其他的文件或目录有很大的差别，自然也就具有与众不同的特性。图 4-1 展示了数据卷与容器的关系。

首先，数据卷中的数据并不继承于镜像，也不在联合文件系统临时层所管理的范围内，所以镜像层的写时复制机制不会作用于数据卷中的数据，而这些数据也不会被 docker commit 提交到新的镜像之中。因为文件的操作不是在沙盒环境中进行的，而是直接作用于宿主机内真实的硬盘 IO 中，所以我们对数据卷中数据的操作会马上产生效果，并且操作速度也要比对临时层中数据的操作来得快。



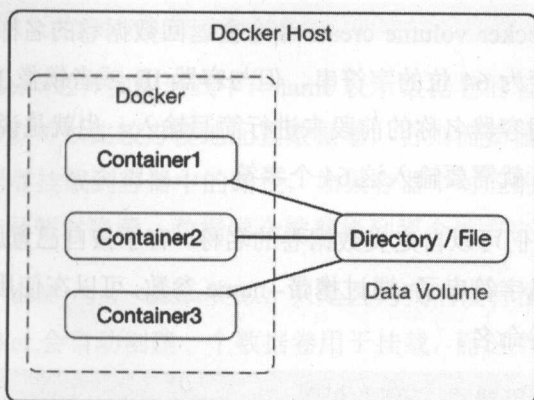


图 4-1 数据卷与容器的关系

其次，数据卷不依赖于容器，它的生命周期也不受容器的控制，所以，我们能够安全地存储文件到数据卷中。也正是因为数据卷独立于容器之外，所以多个容器可以共享同一个数据卷。通过数据卷，我们就可以实现在容器之间进行文件数据的共享了。

数据卷为容器提供了持续稳定的存储空间，数据卷中文件的更新，也不会影响到使用它的容器。容器与数据卷的关系，更像是在 Linux 系统下对文件或目录进行的 mount 操作。

### 4.1.3 创建数据卷

创建数据卷的方式有很多种，其中比较常用的是创建容器时一同创建数据卷。在使用 `docker create` 或 `docker run` 创建容器时，可以通过 `-v` 参数向容器中挂载一个数据卷，Docker 会自动创建这个数据卷。

```
$ sudo docker create --name web -v /html nginx
```

这样，我们就为 Web 容器创建了一个路径为 `/html` 的数据卷，所有对 `/html` 中的文件的访问与操作，都是对数据卷中的数据进行访问和操作。通常情况下，我们还需要同时挂载多个数据卷到同一个容器。多次使用 `-v` 参数，就可以分别挂载这些需要的数据卷了。

```
$ sudo docker create --name web -v /html -v /var/log/nginx nginx
```

还可以使用 Docker CLI 中专有的创建数据卷的 `docker volume create` 命令来创建数据卷。

```
$ sudo docker volume create
9357d69253e5e8137df0b4b8dafc7455ea71f19cb8009d68ef72050eecbc9646
```

创建数据卷后，`docker volume create` 命令会返回数据卷的名称。默认的数据卷名称是一个随机产生的长度为 64 位的字符串。但与容器 ID 或者镜像 ID 不同的是，在绝大多数情况下，不能使用容器名称的前段来进行简写输入。也就是说，一旦输入命令时需要使用数据卷的名称，就需要输入这 64 个字符。

为了使用方便，我们可以自定义数据卷的名称，有了按自己意愿定义的数据卷名称，就不用记录冗长的随机字符串了。通过携带 `--name` 参数，可以在使用 `docker volume create` 命令时为创建的数据卷命名。

```
$ sudo docker volume create --name html
html
```

创建数据卷之后，可以通过 `docker volume inspect` 命令查看数据卷的基本信息。

```
$ sudo docker volume inspect html
[
  {
    "Name": "html",
    "Driver": "local",
    "Mountpoint": "/var/lib/docker/volumes/html/_data",
    "Labels": {},
    "Scope": "local"
  }
]
```

我们可以发现，数据卷其实就是放置在数据主机内的一个特殊目录，只是 Docker 将其封装在容器中进行展示。通过 Docker 的封装，程序在容器中就无法判断数据卷到底从何处挂载，也觉察不出数据卷与其他文件或目录有任何区别，它们甚至不知道容器中哪些目录是数据卷，哪些又不是。正是 Docker 为数据卷设计的良好封装性，使迁移过程结束后，程序不会因为数据卷在本地环境中的改变，而不能正常运行。

#### 4.1.4 挂载数据卷

通过创建容器而创建的数据卷，会自动挂载到容器之中，成为容器中目录树的一部分。对于通过 `-v` 参数指定的，并且已经存在于宿主机中的数据卷，Docker 也会找到它们并进行挂载。

对于那些通过 `docker volume create` 命令创建的数据卷，我们可以在使用 `docker create` 或 `docker run` 命令时，采用 `-v <name>:<dest>` 的形式来实现数据卷的挂载。

```
$ sudo docker create --name web -v html:/html nginx
```

在 `-v <name>:<dest>` 这种参数形式中, `name` 表示数据卷的名称。因为我们使用了定义的数据卷名称, 所以可以比较方便地配置数据卷, 否则需要输入冗长的数据卷随机名称。而 `dest` 表示数据卷挂载到容器中的路径, 如果容器中对应的目录已经存在, 那么挂载操作会把原有的目录暂时隐藏, 数据卷会被替换到那个地方去。

其实 `-v <name>:<dest>` 与 `-v <dest>` 类似, 只是因为我们没有在 `-v <dest>` 这种形式中给出数据卷, 所以 Docker 会自动创建一个数据卷用于挂载, 而这个创建的数据卷没有直接展示给我们。

既然 Docker 能够将这些宿主机中的特殊目录挂载到容器里作为数据卷, 那自然也能将宿主机中其他已经存在的目录挂载到容器中当作数据卷。在创建容器时, 可以使用 `-v <src>:<dest>` 这种参数形式, 挂载宿主机中的目录到容器中。

```
$ sudo docker create --name web /var/html:/html -v /var/log/nginx:/var/log/nginx nginx
```

我们除了挂载宿主机的目录作为数据卷, 还可以挂载单个文件到容器之中。不过更推荐以目录的形式挂载数据卷, 因为如果直接挂载文件到容器里, 当我们在容器中使用 Vim 或其他文件编辑工具对文件进行修改时, 可能会造成文件 `inode(inode)` 信息的改变。而这些改变并不在容器内外同步, 这就会造成一些意想不到的错误。挂载文件更适用于文件不被修改的场景, 例如挂载配置文件到容器中。

因为 `-v <src>:<dest>` 与 `-v <name>:<dest>` 两种参数传入形式是一致的, 为了保证不出现歧义, 在使用 `-v` 挂载宿主机目录或文件时, 必须使用目录或文件的绝对路径。

使用挂载宿主机目录或文件成为数据卷的方式, 可以非常方便地在容器内部与外部之间共享文件。特别是将配置、代码类可能需要临时修改的文件, 通过主机目录挂载的方式放置到容器中时, 修改结果会马上体现到容器里, 让修改文件变得更加简单。

挂载宿主机目录或文件到容器中时, 我们还可以对这些数据的可访问性做一定的限制。当我们希望容器中的程序只能读取数据卷的内容, 而不能对数据卷中的数据进行修改的时候, 可以在挂载参数的后面加上 `:ro` 三个字符, 达到只读挂载数据卷的目的。例如, 在一个 Web 服务容器中, 我们希望 HTML 代码目录和其内容是只读的, 而日志目录则应该是可读写的。

```
$ sudo docker create --name web /var/html:/html:ro -v /var/log/nginx:/var/log/nginx nginx
```

只读挂载能在安全性上作出一定的贡献。只读挂载使攻击者入侵到容器中时, 无法



对程序数据进行修改。同时，由于容器对应用之间的隔离、对文件系统的隔离、对物理主机的隔离，攻击者无法进行其他操作。他就好像被封闭在了容器的密闭空间中，找不到出路。

### 4.1.5 删除数据卷

因为数据卷是脱离容器而存在的，虽然我们是通过创建容器的命令创建的数据卷，但这些数据卷依然不会因为容器的停止和删除被销毁。因此，我们才能让数据卷同时提供给不同的容器，并做到数据卷的重复使用。

数据卷不会因为容器的删除而被删除，已经被删除的容器所挂载的数据卷仍存在于宿主机中，并保存着所有数据、占据着存储空间。我们可以通过 Docker 提供的数据卷管理命令，查看宿主机中所有存在的数据卷。

```
$sudo docker volume list
```

DRIVER	VOLUME NAME
local	8bfecd31c395546407c09674eb6d681bbdb1ae368667ad643f0a9bbd1f31fd20
local	9270919083cffb470d1e70dc65eb3c0e5a56b84a5db33304ec812cd7bb60cb2c
local	9357d69253e5e8137df0b4b8dafc7455ea71f19cb8009d68ef72050eecbc9646
local	97f609fb78ea2220c852c7769128eb39699a01d63c49255ef77c184a06878e0a
local	987cd8265f84eb2163b4f438da6a0d1bf04fd9f8f230753b8d921ec11d5de918
local	9e249ef406e83256f6fb41657249b91a92216e04440f431663dacb281cd6cd35
local	a1688a6f7a5ad70289b7b4f7dcb94304f81314b85d580e158590e9d4abe65d44
local	badefd0f2b13ae2f044090bf8098f91499cfafaa9df4724cb3f5fbd426f88f6e
local	c2cade37380dc32b403959d681f3ac26ce4931fc5ab9090bb6ce19e50778f676
local	d83368befc58f7568d102dd153b01f27578d463ebffc2747ab589d467be4660a
local	f37d4df83d32c66ac4cef5fa2818d177d5ba4baf6d959f5ac2c082230a40432a
local	fdec34b5a76fda4d6df4ca5a0a42b153ba7d74610adc8f2726fb5a60a825f65b
local	ffa1fb855ce7875682cc6055e8777152719dbc535db44b5e6f2fe59b9ded177d
local	html

我们可以使用数据卷管理命令中的删除数据卷命令，来删除这些被列出的存在于宿主机中的数据卷。

```
$ sudo docker volume rm badefd0f2b13ae2f044090bf8098f91499cfafaa9df4724cb3f5fbd426f88f6e
badefd0f2b13ae2f044090bf8098f91499cfafaa9df4724cb3f5fbd426f88f6e
```

使用 `docker volume list` 命令会列出宿主机中的数据卷，既有未被容器使用的废弃数据卷，也有正在被容器使用的数据卷。特别是对于那些并未设置数据卷名称，只采用随

机名称的数据卷，我们很难判断它们是否正在被数据卷使用。所以，我们在使用 `docker volume rm` 命令时需要特别小心，不要误删正在被容器使用的数据卷，以免引起错误。

若要删除数据卷，我们更推荐另外一种方式——随容器删除。我们使用 `docker rm` 命令删除容器时，可以带入 `-v` 参数一同删除容器使用的数据卷。

```
$ sudo docker rm -v web
```

使用随容器删除数据卷的方式，并不是总会删除数据卷。因为数据卷是脱离容器存在的，而我们也能够把一个数据卷同时挂载到不同的容器中，所以在使用随容器删除数据卷的方式时，Docker 会先检查容器中挂载的数据卷是否正在被其他容器使用，只有数据卷没有在其他容器中使用，才会将这个数据卷从本地文件系统中删除。

另外，拥有名称的数据卷也不会因为使用 `-v` 参数而被删除。也就是说，随容器删除数据卷这个功能，针对的是直接在创建容器时创建的数据卷。

## 4.2 数据卷容器

若要更好地在容器之间共享数据卷，可以使用数据卷容器来实现。通过数据卷容器，我们可以更轻松地将数据卷进行归类 and 汇总，也能更好地管理容器与数据卷之间的关系，并且可以更加合理地控制数据卷的生命周期。

### 4.2.1 关于数据卷容器

随容器创建的数据卷最好也随容器删除。而需要长期存储数据的数据卷，特别是用于持久化保存数据的数据卷，可以通过挂载宿主机目录的方式来实现。不过这种方式仍然存在弊端——容易破坏 Docker 的统一性。为了解决这个问题，我们可以使用数据卷容器来管理数据卷。

数据卷容器是专门用于存放数据卷的容器，我们在其他的容器中使用数据卷时，就不再把宿主机的目录当作数据卷进行挂载，而是从数据卷容器中将数据卷挂载。如图 4-2 所示，容器通过数据卷容器衔接到数据卷上。

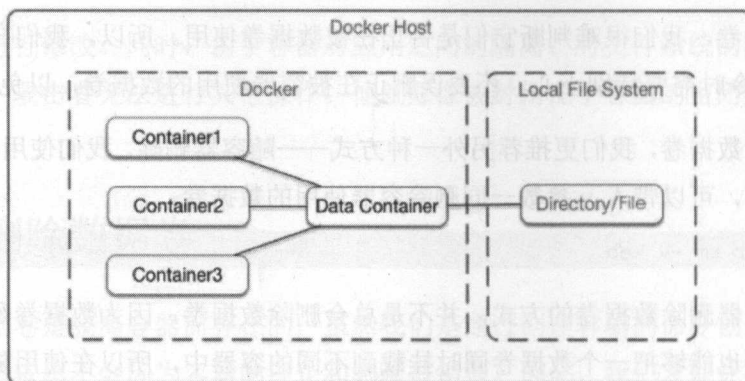


图 4-2 数据卷容器

数据卷是独立于容器而存在的，那么，数据卷容器又是如何管理数据卷的呢？其实，数据卷容器并没有直接管理或控制数据卷，它只是其他容器使用数据卷的桥梁。我们知道，使用 `-v` 参数删除数据卷时，Docker 会保证此数据卷没有被其他容器所使用。而当我们使用数据卷容器进行数据卷的挂载时，虽然数据卷被其他容器所引用和使用，而其他容器在删除时也能尝试对数据卷进行删除，但是由于数据卷仍然被数据卷容器所引用，Docker 就不会删除这个数据卷以及其中的数据。这样就有力地保证了数据卷的安全，我们也能通过数据卷容器来清楚地了解数据卷的状态，而不需要在 `docker volume list` 的结果中逐个核对。

因为数据卷容器只是其他容器与数据卷连接的桥梁，而数据卷也可以独立于容器存在。所以我们在其他容器中通过数据卷容器来访问数据卷时，并不需要保证数据卷容器必须在运行的状态。也就是说，其他容器只是利用数据卷容器所给出的数据卷信息，在 Docker 中找到对应的数据卷，并不是一直通过数据卷容器来访问数据卷，因此数据卷容器的运行与否，并不影响其他容器对数据卷的使用。

## 4.2.2 创建数据卷容器

数据卷容器的创建与普通容器的创建方法没有差别，都是通过 `docker create` 或 `docker run` 命令来创建。因为使用数据卷容器时无须保证数据卷容器处于运行状态，所以我们通常使用 `docker create` 命令创建数据卷容器。不让数据卷容器运行起来，可以减少其对宿主机性能的影响。

```
$ sudo docker create --name data -v /html ubuntu
```

在创建数据卷容器时，不要忘记使用 `-v` 参数来建立数据卷容器所使用的数据卷。在



同一个数据卷容器中，也可以绑定多个数据卷。不过，为了能够更准确地管理数据卷，最好还是使用不同的数据卷容器来存放不同的数据卷；或者将数据卷进行分类，分别放在不同的数据卷容器中。

创建数据卷容器时所使用的数据卷目录，会在其他容器连接到此数据卷容器时，作为对应访问数据卷的目录。所以在设置数据卷目录时，要按容器实际使用的数据卷的目录来配置。

对于容器中所使用的数据卷，我们可以通过 `docker inspect` 命令来查看。自然的，数据卷容器里绑定的数据卷，也可以通过 `docker inspect` 命令查看。

```
$ sudo docker inspect data
[
  {
    ...
    "Mounts": [
      {
        "Name": "fcfe4ed4082b4e023b3f6b1f91dbad04516658cb3f236a43786f576216d2ee56",
        "Source": "/var/lib/docker/volumes/fcfe4ed4082b4e023b3f6b1f91dbad04516658cb3f236a43786f576216d2ee56/_data",
        "Destination": "/html",
        "Driver": "local",
        "Mode": "",
        "RW": true,
        "Propagation": ""
      }
    ],
    ...
  }
]
```

### 4.2.3 连接数据卷容器

数据卷容器只是连接其他容器与数据卷的桥梁，其他容器如何通过数据卷容器连接到自己所需要的数据卷，是数据卷容器使用中非常重要的环节。

在创建容器时，通过 `--volumes-from` 参数可以挂载指定数据卷中所有的数据卷。

```
$ sudo docker run -d --name web --volumes-from data nginx
```

当我们在创建容器时传入 `--volumes-from` 参数时，Docker 会查询到给出的数据卷容

器中的数据卷，并将它们挂载到正在创建的容器中。这些数据卷的挂载点，会使用创建数据卷容器时所指定的数据卷挂载目录。

我们还可以多次使用 `--volumes-from` 参数，实现对多个不同的数据卷容器的同时使用，这些数据卷容器中所有的数据卷，都会被挂载到新的容器中。

```
$ sudo docker run -d --name web --volumes-from data --volumes-from logs nginx
```

当多个容器同时使用了相同的数据卷容器所提供的数据卷时，它们所挂载目录中的文件是相通的。也就是说，在一个容器里对挂载目录中的文件进行操作，也会马上反映到另外一个容器的对应文件上。这就让我们能够在不同的容器中，实现对相同文件的即时访问和操作，达到容器间共享文件数据的目的。

## 4.2.4 数据卷的迁移

数据的备份或导出是保证数据安全最简单直接的方式。在对 Docker 进行运维的过程中，自然也离不开对数据卷中数据的导出或备份工作。

虽然是对容器中数据卷的备份，但并不是通过 Docker 自带的方式来实现的，而是利用 Tar 存档管理工具来进行的。因为整个备份操作是针对数据卷进行的，所以我们称其为数据卷的导出备份。

因为数据卷导出并不利用 Docker 自身提供的方法，而需要导出的数据卷往往又不以宿主机目录形式进行挂载。所以要让这些数据被 tar 命令访问到，必须先进入到容器中去，在容器中利用 tar 命令将数据卷目录进行打包。然后再将打包的数据从容器中移动到容器外进行保存。而这又需要我们挂载一个容器外的目录，用来接收已经打包好的备份数据。

已经启动的容器是无法再挂载新的外部目录进去的，所以即使我们能进入容器打包数据，也很难将数据转移到宿主机中。而容器中的数据卷则不同，因为数据卷是可以被多个容器挂载的，我们可以通过新建容器挂载宿主机中的目录，并共享需要导出的容器里的数据卷，这样就可以很容易地实现数据的导出了。

既然导出数据需要利用数据卷能够在容器中共享的特性，自然我们就要选择最佳的容器间共享数据卷的方式——数据卷容器，方便导入/导出也是使用数据卷容器的优势之一。

要导出数据，需要创建新的容器，并将其连接到持有我们需要导出数据卷的数据卷容器上。容器创建和运行后，就可以进入容器执行打包命令，并将导出数据放置到挂载了宿主机目录的地方。

```
$ sudo docker run -it --volumes-from data -v $(pwd):/backup --name exporter --rm ubuntu
/bin/bash
root@0cf3e7210eb0:/# tar cf /backup/data.tar /data
root@0cf3e7210eb0:/# exit
```

我们可以通过进入容器的方法执行打包数据的命令，也可以在创建容器时就把容器的启动命令设置为打包数据的命令。

```
$ sudo docker run -it --volumes-from data -v $(pwd):/backup --rm ubuntu tar cf
/backup/data.tar /data
```

当容器启动时，tar 程序就会把数据卷文件夹中的数据打包到 data.tar 中，由于 data.tar 存放的/backup 目录是我们从宿主机挂载的，所以这个文件已经到了宿主机中。而打包程序结束，容器也会自动停止。因为我们使用了--rm 参数，所以容器停止后也会自动删除。这样，虽然整个导出过程历经了创建容器—运行命令—关闭和删除容器的过程，但是因为容器的轻量级，所以完成整个过程非常快，我们甚至可以把这条命令当成专用于导出数据卷的命令。

要恢复导出的数据，只要把导出的过程逆向执行一遍即可。

恢复数据之前，依然要创建一个新的容器，挂载上宿主机中存在备份数据的目录，并连接到包含目标数据卷的数据卷容器上。然后运行并进入容器，在容器中执行解包命令，把导出的数据放置到目标数据卷里。

```
$ sudo docker run -it --volumes-from data -v $(pwd):/backup --name importer --rm ubuntu
/bin/bash
root@c6f2313c77eb:/# tar xf /backup/data.tar
root@c6f2313c77eb:/# exit
```

和导出数据时的操作一样，我们也可以不进入容器，把解包命令当成容器的启动命令，直接实现数据导入。

```
$ sudo docker run -it --volumes-from data -v $(pwd):/backup --rm ubuntu tar xf
/backup/data.tar
```

我们使用 tar 命令时都在根目录执行，而压缩路径则使用了绝对路径，这样能更方便地在备份数据中记录数据卷所在的目录结构，也方便恢复。在实践中，大家可以根据需要自己定义备份压缩的参数及路径，以更好地满足实际需求。



## 4.3 网络基础

网络是互联网时代下，任何一款应用程序都离不开的数据交换通道。Docker 作为分布式部署的利器，自然也为容器提供了强大的网络支持。那么，Docker 为网络模块做了怎样的实现和优化？我们就在这一节中进行简单的介绍。

### 4.3.1 网络简介

大量的基于分布式云计算的互联网应用服务，都离不开通过网络对外向用户提供服务，而对内也需要进行不同服务组件，或者说是微服务之间的信息交流。Docker 作为云计算领域应用部署的佼佼者，对容器使用的网络也提供了强大的支持。

容器技术的特点就是隔离性，网络作为计算机重要的资源之一，自然也在容器隔离的范围之内。Docker 通过 Network Namespace 的方式，为每一个容器建立了独立的网络，形成了完全与宿主机隔离的环境。Network Namespace 与我们之前谈到的用于隔离应用运行环境的 Namespace 技术相似，是用来隔离应用运行的网络环境的。它能让应用运行所在的网络环境完全虚拟出来，并能提供独立的网络设备、IP 协议栈、IP 路由表、防火墙、/proc/net 目录、/sys/class/net 目录、端口套接字等的支持。运行在 Network Namespace 下的应用，就如同拥有了单独的网络硬件和单独的网络配置。

默认情况下，Docker 启动时会在宿主机上架设一个名为 docker0 的虚拟网络，用来连接宿主机与容器。既然容器的网络是以独立隔离的形式存在的，那么 Docker 又是如何利用 docker0 来实现容器与宿主机，乃至外界网络之间的相互通信的呢？

容器启动时，Docker 会把容器内通过 Network Namespace 建立的独立网络，通过 Veth Pair 连接到 docker0 所在的虚拟网络上。Veth Pair 是专门用于虚拟网卡间通信的通道，它就像普通的数据管道一样，将容器中的网络数据包传递到 docker0 上，也将 docker0 发出的数据包传递到容器中。通过 Veth Pair 在本来已经隔离的 Network Namespace 打出了一个洞，把容器内部网络与容器外部网络连接起来。

如图 4-3 所示，当容器内部的网络通过 Veth Pair 连接到 docker0 上时，它就与其他连接到 docker0 上的容器同属于一个子网中，而宿主机也通过虚拟网卡连接到了 docker0 上。也就是说，宿主机与容器间通过 docker0 达到了互相访问的目的。

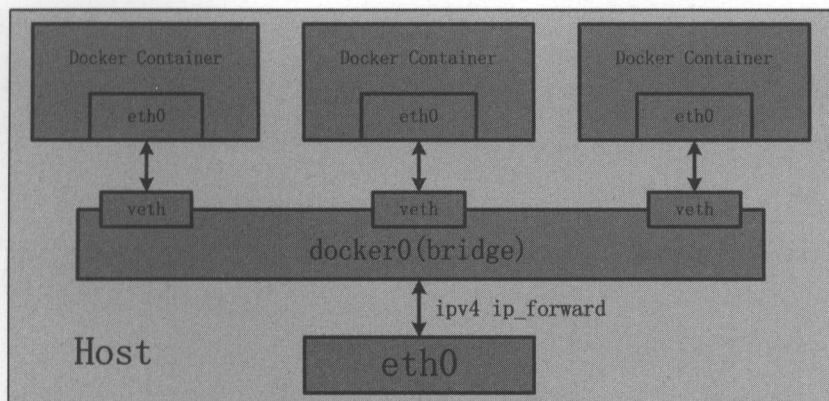


图 4-3 容器的网络拓扑

当然，这种容器与宿主机及容器之间，通过建立 `docker0` 子网来互相访问的方式，只是 Docker 网络模块提供的默认连接方式，Docker 还提供了更丰富的网络配置形式和网络实现方式。本章所探讨的网络，都基于 `docker0` 这个 Docker 默认实现的虚拟子网，对于更丰富的网络搭建和使用方式，我们在后面会专门进行讲解。

### 4.3.2 查看网络配置

要了解 Docker 的网络，自然要从了解容器的网络状态开始。每个容器都运行在隔离的网络空间里，那么我们如何去了解容器所运行的这个独立的网络空间呢？此时就可以使用 `docker inspect` 命令，前面我们讲解过利用它可以查看容器的信息，如容器运行状态、数据卷挂载都会出现在它的结果中。当然，关于容器网络方面的信息，也可以通过 `docker inspect` 命令来查看。

```
$ sudo docker inspect web
```

我们在使用 `docker inspect` 命令查询出的结果中可以找到 `NetworkSettings` 字段。

```
...
"NetworkSettings": {
  "Bridge": "",
  "SandboxID": "0592db95f17821fe778eb90cfb533882e77515ce2305b369500b554a83a7546e",
  "HairpinMode": false,
  "LinkLocalIPv6Address": "",
  "LinkLocalIPv6PrefixLen": 0,
  "Ports": {
    "443/tcp": [
      {
        "HostIp": "0.0.0.0",
```

```

        "HostPort": "443"
    },
    ],
    "80/tcp": [
        {
            "HostIp": "0.0.0.0",
            "HostPort": "80"
        }
    ]
},
"SandboxKey": "/var/run/docker/netns/0592db95f178",
"SecondaryIPAddresses": null,
"SecondaryIPv6Addresses": null,
"EndpointID": "ba6b2c9d2f6d33985ea46f50a182c96cb349f8afa7889bdb395b822f6fc9b325",
"Gateway": "172.17.0.1",
"GlobalIPv6Address": "",
"GlobalIPv6PrefixLen": 0,
"IPAddress": "172.17.0.2",
"IPPrefixLen": 16,
"IPv6Gateway": "",
"MacAddress": "02:42:ac:11:00:02",
"Networks": {
    "bridge": {
        "IPAMConfig": null,
        "Links": null,
        "Aliases": null,
        "NetworkID": "f44002fe7a5db55aca977948676bd5d395d924136f1e6f165403002bfb9835af",
        "EndpointID": "ba6b2c9d2f6d33985ea46f50a182c96cb349f8afa7889bdb395b822f6fc9b325",
        "Gateway": "172.17.0.1",
        "IPAddress": "172.17.0.2",
        "IPPrefixLen": 16,
        "IPv6Gateway": "",
        "GlobalIPv6Address": "",
        "GlobalIPv6PrefixLen": 0,
        "MacAddress": "02:42:ac:11:00:02"
    }
}
}
...

```

在网络信息中，我们也可以看到容器被 Network Namespace 所分配的硬件信息，如 Mac 地址；也可以发现容器在 docker0 子网中分配的 IP 地址、网关地址、网桥配置等。



很多时候，我们需要了解容器运行的网络状态，通过这些来掌控容器的情况，并在需要的时候利用这些信息解决容器出现的问题。通过 `docker inspect` 命令了解容器的网络信息，为我们使用容器提供了很好的帮助。

## 4.4 网络访问

在 Docker 容器中，网络资源也同其他计算机资源一样，与宿主机隔离开了。那么如何让容器中的应用程序访问到网络中的数据，又如何让其他程序通过互联网访问到容器中所提供的服务？在这一节中，我们就讲解如何使用控制容器网络的命令。

### 4.4.1 宿主机端口映射

容器的运行处于独立的网络空间，即使有 `docker0` 子网作为网桥存在，也只是让容器与宿主机有了网络沟通渠道而已。也就是说，容器与外部网络之间仍然处于不能互通的状态下。Docker 作为分布式部署的利器，运行在其中的程序自然需要与容器外部的网络进行数据交换。

在 Docker 默认的 `docker0` 网络实现中，Docker 已经为容器完成了外网访问的相应配置。容器中的程序如果需要访问外网的主机，请求它们的数据，网络的访问可以通过 `docker0` 被转发到宿主机的外网网卡上。所以在容器中，我们可以直接访问宿主机能访问的网络。

虽然容器可以直接对外部网络进行访问，但是外部网络是无法直接访问容器的。因为在默认情况下，外部网络与容器网络进行连接，必须通过 `docker0` 的网关，但是由于可能有多个容器同时存在，所以这个网络中继无法对所有容器同时进行转发。解决这个问题的方案有很多，如在简单的 Docker 网络配置命令和参数中，采用了端口映射的方式来处理这个问题。对于我们想让外部网络访问的容器，可以向 Docker 提供接受访问的端口，让实现与宿主机的端口绑定。

在创建容器时，可以通过携带 `-P` 参数将容器的端口绑定到宿主机的端口上。

```
$ sudo docker run -d -P nginx
```

通过 `-P` 参数进行的绑定属于随机绑定，Docker 会在宿主机上寻找可用的端口，绑定到容器的端口上。也就是说，相同镜像创建出的不同容器，即使容器中的相同端口，在

宿主机上绑定的端口也不一定是相同的。我们可以在列出容器命令的列表中，看到容器的端口实际绑定到了宿主机的哪个端口上。

```
$ sudo docker ps -l
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
PORTS			NAMES	
bfe5d65801a2	nginx	"nginx -g 'daemon off'"	4 seconds ago	Up 3 seconds
0.0.0.0:32769->80/tcp, 0.0.0.0:32768->443/tcp			jovial_kalam	

宿主机的端口不足以容纳下容器所有的端口，当然 Docker 也不会让宿主机映射容器所有的端口。能被 Docker 映射到宿主机上的容器端口，是被创建容器的镜像预先定义好的，这些被定义的端口，通常就是容器中的程序对外提供服务所监听的端口。例如，我们在示例中使用 Nginx 镜像和它所创建的容器，会使用 80 和 443 号端口进行映射，因为在 Nginx 向外提供的 Web 服务中，HTTP 和 HTTPS 分别使用了 80 和 443 号端口作为默认端口。

使用 -P 参数，可以让 Docker 为容器需要暴露的端口，选择空闲的宿主机端口进行绑定。这样可以避免不同容器暴露和映射端口时出现冲突，但每次映射的端口都不能保持一致，不利于外部程序的连接适配。

在大部分情况下，我们还是希望有确定的端口实现宿主机与容器间的绑定，而不是让 Docker 自行寻找可用的端口进行绑定。通过容器创建时传入的 -p 参数，就能实现确定的端口映射设置。需要注意的是，两种设置端口映射的参数非常相似，直接设置全部端口并随机分配映射端口的参数是 -P (大写字母)，而精确设置单一端口映射的参数是 -p (小写字母)。指定映射参数的使用形式是 -p <hport>:<cport>，在 -p 后分别传入宿主机希望被使用于映射的端口，以及被映射的容器端口即可。

```
$ sudo docker run -d -p 80:80 -p 443:443 nginx
```

```
$ sudo docker ps -l
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
PORTS			NAMES	
012f8a98a213	nginx	"nginx -g 'daemon off'"	9 seconds ago	Up 8 seconds
0.0.0.0:80->80/tcp, 0.0.0.0:443->443/tcp			suspicious_jennings	

因为 -p 参数是用于确定一条端口映射关系的，当容器中有多个端口需要进行映射时，要将每一个需要映射的宿主机和容器端口通过 -p 参数传入。

有时，我们不但需要指定宿主机上用于映射的端口，还希望对可访问的外部主机进行限制。通过 -p <ip>:<hport>:<cport> 这种参数形式，可以附加映射端口针对地址访问监听的限制。

```
$ sudo docker run -d -p 116.211.1.106:80:80 -p 116.211.1.106:443:443 nginx
$ sudo docker ps -l
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
f8c90fefb69b	nginx	"nginx -g 'daemon off'"	6 seconds ago	Up 5 seconds
127.0.0.1:80->80/tcp, 127.0.0.1:443->443/tcp		ecstatic_bell		

这样设置端口映射后，容器就会只收到来自主机 IP 为 116.211.1.106 的请求。

对于需要混合使用确定分配和随机分配映射关系的场景，也可以利用 `-p` 参数来完成指定端口的随机映射。将 `-p` 参数所需的宿主机端口省略，**Docker** 就会让这个容器端口映射到宿主机上随机分配的空闲端口上了。

```
$ sudo docker run -d -p 80:80 -p 443 nginx
$ sudo docker ps -l
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
aa39ebe95b52	nginx	"nginx -g 'daemon off'"	3 seconds ago	Up 2 seconds
0.0.0.0:80->80/tcp, 0.0.0.0:32768->443/tcp		condescending_hawking		

对于既要指定监听地址又要使用随机分配端口的场景，可以使用如下形式来配置。

```
$ sudo docker run -d -p 183.6.214.14::80 -p 443 nginx
$ sudo docker ps -l
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
c43e80adf391	nginx	"nginx -g 'daemon off'"	3 seconds ago	Up 2 seconds
183.6.214.14:32769->80/tcp, 0.0.0.0:32768->443/tcp		suspicious_kalam		

使用 **Docker** 运行的大多数程序都不是孤立存在的，它们或多或少都需要与外部程序进行信息交换，而大部分交换信息的渠道又来自网络。所以说，进行容器端口与宿主机端口的映射，是使用 **Docker** 网络需要掌握的最基本的技术。

## 4.4.2 容器连接

端口映射是在默认情况下，容器突破 `docker0` 子网，实现将外部网络对特定端口的访问转接到容器内的端口上，使容器内运行的应用程序，能够处理来自宿主机外部的程序，并通过网络发送处理请求。

有时，一个容器中运行的应用程序，也需要与运行在另外一个容器中的应用程序，通过网络进行数据交换。此时，就需要通过容器连接来完成容器与容器之间的网络连接了。



要设置容器间通信，我们可以通过在创建容器时携带`--link`参数来实现，参数的使用形式是`--link <container>`。使用这个参数创建的容器，会打开对被连接容器的网络访问。在新的容器中，我们就能通过被连接容器暴露的端口，连接和请求被连接容器中的应用程序，并获得它们提供的数据。

这里我们创建一个 MySQL 容器，并让一个 Web 服务容器连接到它。

```
$ sudo docker run -d --name mysql mysql
$ sudo docker run -d -p 80:80 -p 443:443 --name web --link mysql nginx
```

设置容器间通信的参数，只需要指定被连接的容器，并不需要指明被连接容器的端口，在创建和启动被连接容器时，也不需要通过`-P`或`-p`参数来映射端口。这就保证了被连接容器的端口只在容器间的通信中使用，不会被暴露在外网中，也不会被其他容器访问到。

当我们在容器创建时使用了连接容器的参数，并且被连接的容器也正常运行起来了，那么我们就能够在容器中，使用`--link`参数指定的名称访问到被连接的容器了。例如，在上例的 MySQL 服务中，我们可以在 Nginx 容器中需要访问 MySQL 的地方，将服务的主机名 (Host) 设置成 `mysql`，这样就能让这些程序访问到 MySQL 容器中的 MySQL 服务了。因为网络数据交换和文件数据交换不同，网络连接时必须保证两个互相连接的程序都处于运行的状态。也就是说，连接和被连接的容器都要处于运行状态，才能保证容器与容器之间的网络通信正常建立。

在连接运行在其他容器中的程序时，并没有使用被连接容器分配在 Docker 子网中的 IP，而是使用了容器的名称作为连接的主机名，这种实现方式是 Docker 做的一层封装。因为在常见的场景中，不同的主机、不同状态的 Docker 配置，为容器分配的 IP 往往是不同的。如果直接使用 IP，就需要在迁移时再对程序或程序的配置进行修改，做适配环境的工作，这就大大增加了程序重复部署过程中的工作量。若通过 Docker 进行容器名与 IP 的映射，就能直接使用容器名进行访问，可以有效避免迁移或重新创建所带来的 IP 改变造成的问题。

在某些情况下，被连接容器的名称可能与连接容器内的某些配置重名，导致使用过程中出现二义性并引发冲突。对于这种情况，Docker 支持容器间使用别名进行连接的方式。在指定容器连接时，可以使用`--link <name>:<alias>`的方式来设置被连接的容器，以及在新容器中访问被连接容器时使用的别名。

```
$ sudo docker run -d --name mysql mysql
$ sudo docker run -d -p 80:80 -p 443:443 --name web --link mysql:db nginx
```

通过这种方式定义后，我们在 Nginx 容器中访问 MySQL 容器的程序时，就可以使用 db 作为访问时的主机名了。

Docker 内部主要通过两种方式为容器公开连接信息，其中一种是环境变量。在容器中，我们使用 env 命令就能看到这个容器当前的环境变量，Docker 会自动配置一些环境变量，我们也可以通过创建容器时携带--env 参数来自定义环境变量。我们在运行的 Web 服务容器中查看容器当前的环境变量：

```
$ sudo docker exec -it web /bin/bash
root@dc6b4b4f0981:/# env
```

可以找到与容器连接相关的环境变量信息：

```
...
DB_NAME=/web/db
DB_PORT=tcp://172.17.0.2:3306
DB_PORT_3306_TCP_PORT=3306
DB_PORT_3306_TCP_PROTO=tcp
...
```

这里清晰地表明了被共享的 MySQL 容器的内网地址及开放的端口。

除了环境变量，MySQL 容器的信息也被添加进了 Nginx 容器的 Hosts 中，我们通过查看 Nginx 容器中的/etc/hosts 文件就能看到：

```
root@dc6b4b4f0981:/# cat /etc/hosts
127.0.0.1 localhost
...
172.17.0.2 db 023b9afe76f1 mysql
172.17.0.3 dc6b4b4f0981
...
```

这里有两条记录，一条是 Nginx 容器自身，使用 Nginx 容器的 ID 作为主机名。另外一条指向的是 MySQL 容器的内网 IP，其可以使用 db、mysql 及 MySQL 容器 ID 作为访问的主机名。这也是 Nginx 容器中运行的程序可以通过 db 这个主机名访问到 MySQL 容器的原因。

一个容器往往需要多个分布在不同容器的服务程序来支持，通过多次使用--link 参数，可以分别指定这些容器间通信的参数信息。

## 4.5 本章小结

数据是程序处理的主要对象，因此任何程序都离不开数据交换的过程。在分布式环境中，文件和网络分散在不同地域空间或不同运行时间下，是程序交换的重要桥梁。对文件和网络数据传输的支持，也是分布式部署程序不可或缺的一部分。

通过对本章的学习，我们认识了 Docker 在对容器的文件与网络数据传递方面的支持，也掌握了数据卷、数据卷容器及 Docker 基础网络架构方面的知识。在数据卷方面，讲解了数据卷的创建、删除等基本操作，并且展示了数据卷如何在容器之间传递数据，还讲解了如何通过数据卷容器管理数据卷及数据卷中数据的导入与导出。在网络方面，简单介绍了 Docker 网络架构，并对基本的容器与容器、容器与外界进行网络连接有了了解。

当然，数据拥有与生俱来的复杂性，Docker 对文件数据与网络数据的支持仍然不够成熟，也并不完美。不过，Docker 的工程师们正努力扩展和优化这方面的功能。相信随着时间的推移，Docker 在数据卷与网络方面的新功能、新特性又会让我们为之震撼。



# 第 5 章

## 制作镜像

在之前的章节中，我们谈到了如何使用 `docker export` 和 `docker import` 命令来导出和导入容器，也介绍了如何使用 `docker commit` 命令来创建容器的提交，而且这两种方式都能持久化我们对容器的修改。在实际工作中，我们不但要对容器的状态进行保存，还需要把导出的容器状态或创建的容器提交共享给其他机器。我们可以通过直接传输导出的容器和提交生成的镜像来完成这个过程，但若遇到结构较为复杂、体积空间比较庞大的导出数据或镜像时，直接传输往往会产生很多意想不到的问题。

为了简化制造镜像的过程，方便在多台主机上共享镜像，Docker 提供了一种通过配置文件创建镜像的方式——使用 `Dockerfile` 构建镜像。这种方式是将制作镜像的操作全部写入到一个文件中，而 `docker build` 命令可以读取这个文件中的所有操作，并根据这些配置创建相应的镜像。`Dockerfile` 让创建镜像的过程变得更加独立和透明，也使整个过程可以轻松容易地往复执行，是构建镜像和进行容器迁移时一个非常优秀的辅助工具。

本章我们将对 `Dockerfile` 进行介绍，了解 `Dockerfile` 中配置的含义和使用方法，并掌握如何使用 `Dockerfile` 构建镜像。

### 5.1 了解 Dockerfile

`Dockerfile` 虽然是 Docker 中构建镜像最简单、最便捷的方式，但它并不晦涩难懂，

其简单的规则和精简的语法，可以让我们很快上手实践。要掌握如何在 Docker 中使用 Dockerfile 去构建镜像，首先要了解 Dockerfile。

### 5.1.1 Dockerfile 简介

Dockerfile 是由 Docker 提供的进行镜像自动化构建的配置文件，包含了所有用于构建镜像所执行的命令。通过 Dockerfile 可以清晰明确地指定 Docker 在制作镜像过程中执行的操作，也可以轻松地通过迁移 Dockerfile 到其他机器来实现镜像的迁移。

Dockerfile 虽然不具备扩展名，但其仍然是一个简单的文本文件，我们可以很轻松地通过各种编辑器来建立它。Dockerfile 中的内容主要以两种形式出现：注释行和指令行：

```
# Comment
INSTRUCTION arguments
```

以#开头的文本均为注释行，可以在其中写上我们对 Dockerfile 的解释，以及每行指令代表的意义。

指令行主要分为两部分，行首是 INSTRUCTION，即指令的名称；然后是 arguments，即指令所接收的参数。Dockerfile 提供了很多指令，分别代表在构建镜像的过程中，以及基于镜像生成的容器等场景下需要执行的命令或参数。在 Dockerfile 中，指令是不区分大小写的，也就是说，可以使用小写形式的指令，但为了更清晰地分辨指令和参数，指令一般采用大写形式。

下面是 Dockerfile 中一个注释与指令的示例：

```
# Echo a message
RUN echo 'we are building image.'
```

需要注意的是，在 Dockerfile 中，并非所有以#开头的行都是注释行，有一类特殊的参数是通过以#开头的行来指定的，这类行的基本形式是：

```
# directive=value
```

此类行称为解析指令行 (Parser Directives)，它的主要作用是提供一些解析 Dockerfile 需要使用的参数。解析指令行一般情况下很少被用到，但在某些场合它也能发挥应有的作用。

下面是一个简单的 Redis 镜像的 Dockerfile：

```
FROM alpine:3.4
```

```

# add our user and group first to make sure their IDs get assigned consistently,
regardless of whatever dependencies get added
RUN addgroup -S redis && adduser -S -G redis redis

# grab su-exec for easy step-down from root
RUN apk add --no-cache 'su-exec>=0.2'

ENV REDIS_VERSION 3.2.1
ENV REDIS_DOWNLOAD_URL http://download.redis.io/releases/redis-3.2.1.tar.gz
ENV REDIS_DOWNLOAD_SHA1 26c0fc282369121b4e278523fce122910b65fbbf

# for redis-sentinel see: http://redis.io/topics/sentinel
RUN set -x \
    && apk add --no-cache --virtual .build-deps \
        gcc \
        linux-headers \
        make \
        musl-dev \
        tar \
    && wget -O redis.tar.gz "$REDIS_DOWNLOAD_URL" \
    && echo "$REDIS_DOWNLOAD_SHA1 *redis.tar.gz" | shasum -c - \
    && mkdir -p /usr/src/redis \
    && tar -xzf redis.tar.gz -C /usr/src/redis --strip-components=1 \
    && rm redis.tar.gz \
    && make -C /usr/src/redis \
    && make -C /usr/src/redis install \
    && rm -r /usr/src/redis \
    && apk del .build-deps

RUN mkdir /data && chown redis:redis /data
VOLUME /data
WORKDIR /data

COPY docker-entrypoint.sh /usr/local/bin/
ENTRYPOINT ["docker-entrypoint.sh"]

EXPOSE 6379
CMD [ "redis-server" ]

```

在这个例子中，第一条指令 `FROM alpine:3.4` 中的 `FROM` 指令即我们所要构建的镜像所基于的镜像，通常情况下我们会选择一个系统镜像来布置完整的应用，当然也可以通过基于其他已完成的应用镜像。在接下来的行中，有很多对 Linux 命令行的操作，这



就是我们指定的在构建镜像时所有执行的操作。另外，还有工作目录、设置端口号等指令，这些指令都与我们即将构建的镜像，以及基于这个镜像所运行的容器有着密切的关系。

## 5.1.2 使用 Dockerfile 创建镜像

了解了 Dockerfile 的基本知识之后，要如何通过 Dockerfile 来构建 Docker 镜像呢？Docker CLI 为我们提供了 `docker build` 命令，通过这个命令，我们可以很轻松地按照 Dockerfile 约定的流程构建对应的镜像。

这里我们尝试通过之前提到的 Redis 的 Dockerfile 来构建一个 Redis 镜像：

```
$ sudo docker build ~/Redis
```

需要注意的是，在 `docker build` 命令接收的参数中，提供给 `docker build` 命令的应该是 Dockerfile 所在的目录，而非 Dockerfile 文件本身的路径。`docker build` 命令会自动找到给出的目录下的 Dockerfile 文件，并使用它来生成镜像。如果我们存放 Dockerfile 指令的文件名称并不是 Dockerfile，可以通过携带 `-f` 参数指定要采用的文件名。

整个构建镜像的过程都会输出到屏幕中：

```
Sending build context to Docker daemon 39.42 kB
Step 1 : FROM alpine:3.4
3.4: Pulling from library/alpine
e110a4a17941: Pull complete
Digest: sha256:3dcdb92d7432d56604d4545cbd324b14e647b313626d99b889d0626de158f73a
Status: Downloaded newer image for alpine:3.4
---> 4e38e38c8ce0
Step 2 : RUN addgroup -S redis && adduser -S -G redis redis
---> Running in c77afcf2edcb
---> 2e64218c2be1
Removing intermediate container c77afcf2edcb
Step 3 : RUN apk add --no-cache 'su-exec>=0.2'
---> Running in 253577cd0c91
fetch http://dl-cdn.alpinelinux.org/alpine/v3.4/main/x86_64/APKINDEX.tar.gz
WARNING: Ignoring http://dl-cdn.alpinelinux.org/alpine/v3.4/main/x86_64/APKINDEX.
tar.gz: temporary error (try again later)
fetch http://dl-cdn.alpinelinux.org/alpine/v3.4/community/x86_64/APKINDEX.tar.gz
ERROR: unsatisfiable constraints:
su-exec (missing):
  required by: world[su-exec>=0.2]
```

```

The command '/bin/sh -c apk add --no-cache 'su-exec>=0.2'' returned a non-zero code: 1
[root@iz283zxiseqZ ~]# docker build .
Sending build context to Docker daemon 39.42 kB
Step 1 : FROM alpine:3.4
----> 4e38e38c8ce0
Step 2 : RUN addgroup -S redis && adduser -S -G redis redis
----> Using cache
----> 2e64218c2be1
Step 3 : RUN apk add --no-cache 'su-exec>=0.2'
----> Running in f2264d7d3552
fetch http://dl-cdn.alpinelinux.org/alpine/v3.4/main/x86_64/APKINDEX.tar.gz
fetch http://dl-cdn.alpinelinux.org/alpine/v3.4/community/x86_64/APKINDEX.tar.gz
(1/1) Installing su-exec (0.2-r0)
Executing busybox-1.24.2-r9.trigger
OK: 5 MiB in 12 packages
----> ae12f2ddb999
Removing intermediate container f2264d7d3552
Step 4 : ENV REDIS_VERSION 3.2.1
----> Running in 77f31478160d
----> e6c3c9841fc9
Removing intermediate container 77f31478160d
Step 5 : ENV REDIS_DOWNLOAD_URL http://download.redis.io/releases/redis-3.2.1.tar.gz
----> Running in a92a8fbe2008
----> 03cb809be329
Removing intermediate container a92a8fbe2008
Step 6 : ENV REDIS_DOWNLOAD_SHA1 26c0fc282369121b4e278523fce122910b65fbbf
----> Running in 2d134343d39b
----> bf9377637080
Removing intermediate container 2d134343d39b
Step 7 : RUN set -x && apk add --no-cache --virtual .build-deps gcc
      linux-headers      make      musl-dev      tar&& wget -O redis.tar.gz
"$REDIS_DOWNLOAD_URL"    && echo "$REDIS_DOWNLOAD_SHA1 *redis.tar.gz" | shasum -c
- && mkdir -p /usr/src/redis && tar -xzf redis.tar.gz -C /usr/src/redis
--strip-components=1 && rm redis.tar.gz && make -C /usr/src/redis && make -C
/usr/src/redis install && rm -r /usr/src/redis && apk del .build-deps
----> Running in ede3095c8e8e
fetch http://dl-cdn.alpinelinux.org/alpine/v3.4/main/x86_64/APKINDEX.tar.gz
+ apk add --no-cache --virtual .build-deps gcc linux-headers make musl-dev tar
fetch http://dl-cdn.alpinelinux.org/alpine/v3.4/community/x86_64/APKINDEX.tar.gz
(1/18) Installing binutils-libs (2.26-r0)
(2/18) Installing binutils (2.26-r0)
(3/18) Installing gmp (6.1.0-r0)

```

```
(4/18) Installing isl (0.14.1-r0)
(5/18) Installing libgomp (5.3.0-r0)
(6/18) Installing libatomic (5.3.0-r0)
(7/18) Installing libgcc (5.3.0-r0)
(8/18) Installing pkgconf (0.9.12-r0)
(9/18) Installing pkgconfig (0.25-r1)
(10/18) Installing mpfr3 (3.1.2-r0)
(11/18) Installing mpc1 (1.0.3-r0)
.....
```

Docker 根据 Dockerfile 建立镜像的每一步操作都会生成一层镜像。在建立每层镜像的时候，Docker 都会先查找本地的镜像库中是否含有需要构建的镜像，如果有，就会直接采用这个镜像，可以从构建过程的输出中看到---> Using cache 的字样，表示构建此镜像层时采用了本地已有的镜像。逐个对操作生成单独的镜像层，实现了拆分镜像和对镜像层高效利用的效果。

因为通过 Dockerfile 建立镜像的过程，是按每个操作来生成镜像层的，所以如果想减少镜像层的数量，可以通过&&或 sh 文件合并一些构建过程中的操作，让这些操作只生成一个镜像层。

另外，构建镜像时需要指定镜像的名称及标签，我们可以通过在 docker build 命令后携带-t 或--tag 参数来指定被构建镜像的名称及标签信息：

```
$ sudo docker build -t youmingdot/redis:latest ~/Redis
```

构建操作完成后，我们就能在本地的镜像中找到刚刚构建的镜像了：

```
$ sudo docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
youmingdot/redis	latest	4465e4bcad80	5 weeks ago	185.7 MB

## 5.2 基础指令

基础指令是控制或表述 Dockerfile 整体性质的指令，能够为我们选择基础镜像，也能提供镜像的基本信息。



## 5.2.1 FROM

Docker 的镜像都是在 bootfs 层上实现的，但是我们不必每次构建镜像都从 bootfs 层开始，我们可以直接在其他已经搭建好的镜像上进行修改，FROM 指令就是用来指定我们所要构建的镜像是基于哪个镜像建立的。

作为 Dockerfile 必不可少的最基础的指令，FROM 指令必须作为第一条指令，也就是说，它应该出现在除注释以外的第一行里。不过，在一个 Dockerfile 中是允许出现多个 FROM 指令的，以每个 FROM 指令为界限，都会生成不同的镜像，但是我们还是推荐将生成不同镜像的命令拆分到不同的 Dockerfile 中。

FROM 指令主要有以下几种使用格式：

```
FROM <image>
```

或：

```
FROM <image>:<tag>
```

或：

```
FROM <image>@<digest>
```

使用 FROM 指令时，<tag>和<digest>都是可选的，当不指定这两项时，Docker 会像 docker pull 命令一样选择 latest 这个 tag 的镜像作为基础镜像。

## 5.2.2 MAINTAINER

MAINTAINER 指令的用处是提供镜像的作者信息，其使用格式为：

```
MAINTAINER <name>
```

## 5.3 控制指令

控制指令是 Dockerfile 的核心部分，我们通过控制指令来描述整个镜像的构建过程。

### 5.3.1 RUN

在构建镜像的过程中，我们需要在基础镜像中做很多操作，RUN 指令就是用来给定

这些需要被执行的操作的。由于在构建过程中进行各项操作是不可或缺的过程，可以说 RUN 指令是 Dockerfile 中最常用的指令，甚至没有之一。

RUN 指令有两种使用格式：

```
RUN command param1 param2 ...
```

或：

```
RUN ["executable", "param1", "param2", ...]
```

这两种使用格式代表的意义不同，适用的场景也有区别。

如果以 RUN command param1 param2 ... 这种形式来设置 RUN 指令，在构建镜像时，实际是以 Shell 程序来执行操作。例如，我们定义 RUN mkdir data，实际执行的会是 /bin/sh -c mkdir data。默认情况下，Docker 会选择使用 /bin/sh 作为 Shell 程序，我们可以使用 SHELL 指令切换默认的 Shell 程序。大家可能会觉得使用 RUN command param1 param2... 这种形式来执行命令，有种隔靴搔痒的感觉，执行的命令还要在 Shell 程序中中转一下，但其有一个非常大的优势：支持书写换行。有时指令会比较冗长，如果写在一行中极不方便阅读和排查错误，在 Shell 程序中我们可以使用 “\” 对命令进行拆行，例如：

```
RUN apt-get install -y \  
    autoconf \  
    file \  
    g++ \  
    gcc \  
    libc-dev \  
    make \  
    curl \  
    libxml2 \  
...
```

使用换行连接的方式进行书写可以让命令更清晰。

RUN ["executable", "param1", "param2", ...] 这种形式，是将命令及全部参数逐个传入到方括号中，命令及参数都使用双引号进行引用。使用这种方式来执行命令，可以有效规避在某些基础镜像中没有 Shell 程序，或者用于需要临时切换 Shell 程序的时候：

```
RUN ["/bin/bash", "-c", "echo hello"]
```

Docker 会在一个新的镜像层中执行我们给出的命令，并且在执行完成后提交镜像层，用作 Dockerfile 中下一个指令执行的基础。正是因为 Docker 使用了镜像分层设计，才使

得镜像的提交变得非常“廉价”，即使每次执行 RUN 指令都创建一个新的镜像层，也不会对镜像的体积、性能等造成巨大的影响。

需要注意的是，在使用 RUN 指令时，Docker 判断是否采用缓存构建的依据是给出的指令是否与生成缓存使用的指令一致，也就是说，若本次执行的结果与缓存中不一致，也会采用缓存中的数据而不再执行命令。在某些场合下并不是我们想要的结果，比如使用 RUN apt-get update 时都希望使用最新结果，这时可以使用 docker build 命令加 --no-cache 参数的方式解决这个问题。

### 5.3.2 WORKDIR

WORKDIR 指令用于切换构建过程中的工作目录，如果我们在使用 RUN 指令、ADD 指令、COPY 指令，以及容器运行时才会执行的 CMD 指令、ENTRYPOINT 指令中使用了相对目录，相对目录所基于的就是当前的工作目录。

指令给出工作目录的方式可以是绝对目录，也可以是相对目录。

```
WORKDIR /usr
```

或：

```
WORKDIR local
```

如果给出相对目录，那么在切换工作目录时，会参考当前的工作目录进行。如上例中使用 WORKDIR local 时，实际上切换到的工作目录是 /usr/local，因为切换时参考了之前所定义的工作目录 /usr。

我们也可以在 WORKDIR 指令中使用环境变量：

```
ENV BASEDIR /project
```

```
WORKDIR $BASEDIR/www
```

### 5.3.3 ONBUILD

在 Dockerfile 中，ONBUILD 指令是一条非常特殊的指令，因为它可以携带另外一条指令。ONBUILD 指令允许我们指定另外一条指令，这条指令不会在构建当前镜像时执行，而是在构建其他镜像并使用 FROM 指令把当前的镜像作为基础镜像时执行。简单来说，就是我们能在子镜像构建时运行一些指令。

把我们需要执行的指令放在 ONBUILD 指令之后就能设置一个构建触发器，当其他



Dockerfile 把这个镜像作为基础镜像并进行构建时，执行完 FROM 指令之后，我们通过 ONBUILD 指令设置的指令都将被触发。

ONBUILD INSTRUCTION arguments

可以在 ONBUILD 指定之后放置大部分指令，但某些指令是不允许通过 ONBUILD 指令指定的，比如嵌套使用 ONBUILD ONBUILD。

当我们使用 docker build 命令根据 Dockerfile 构建成镜像时，Dockerfile 不会随镜像本身的迁移而发生变动，也就是说，镜像和生成它的 Dockerfile 不存在绑定关系。那么，我们在生成子镜像时，Docker 如何知道我们指定了哪些触发指令呢？其实，我们通过 ONBUILD 指令给出的所有指令，都会在生成镜像时写入到镜像的特征列表中，可以使用 docker inspect 命令看到镜像的构建命令。

ONBUILD 指令只会在构建子镜像中执行，当子镜像构建完成后，这些指令也随之消失，所以它们不会继承到子镜像或者更后辈的镜像中，也就不会在基于那些镜像的新镜像构建时执行。其实这不难理解，当子镜像构建时，所有通过 ONBUILD 指令给出的指令都已经执行，并且结果也持久化保存到了子镜像中，所以孙辈镜像就不再需要执行这些指令了，因为它们可以直接从基础镜像中获得执行的结果。

## 5.4 引入指令

在很多场合下，我们希望将文件加入到即将构建的镜像中，引入指令就能够帮助我们实现这个目的。

### 5.4.1 ADD

在构建容器的过程中，可能需要将一些软件源码、配置文件、执行脚本等导入到镜像的构建过程，这时可以使用 ADD 指令将文件从外部传递到镜像内部。ADD 指令有以下两种使用形式。

```
ADD <src>... <dest>
```

或：

```
ADD ["<src>",... "<dest>"]
```

两种使用形式并无太大差别，只是 `ADD ["<src>",... "<dest>"]` 这种形式可以规避在文件路径中带有空格的情况。

`ADD` 指令能够将我们在 `<src>` 里指定的文件、目录，乃至通过 URL 指定的远程文件复制到镜像的 `<dest>` 目标路径中。我们可以分别指定多个 `<src>` 文件或目录，也可以使用通配符指定多个文件或目录，如下所示。

```
ADD hom* /mydir/
```

或：

```
ADD hom?.txt /mydir/
```

通配符规则采用的是 Go 语言的文件名称匹配规则，例如使用 `*` 代表任意多个字符，`?` 代表任意一个字符，`[]` 用来配置字符范围等。详细的匹配规则我们可以通过网址 <http://golang.org/pkg/path/filepath#Match> 查询到。

设置导入文件 `<src>` 路径时，需要使用相对路径，这样才能保证移动 `Dockerfile` 和其目录后不需要再进行修改。而 `ADD` 指令中 `<src>` 所指定的相对路径，是相对于镜像的构建路径而言的，也就是我们在 `docker build` 命令中所传入的路径。通常情况下，这个路径就是 `Dockerfile` 所在的路径。另外，`<src>` 所给出的文件还不能脱离 `docker build` 命令所给出的路径，也就是说，不能使用 `../something` 或类似的路径来访问上级路径。

所有被复制进镜像的文件都将放置到 `<dest>` 给出的目录下，这个目录可以是绝对路径，也可以是相对于镜像的工作目录而言的相对路径。镜像的工作目录可以通过 `WORKDIR` 指令来设置，也可以通过 `RUN` 和 `cd` 命令来修改，详细内容在 `WORKDIR` 指令的说明中会提到。被复制进来的文件和目录的所有者 ID 和用户组 ID 都是 0，也就是 `root` 用户，如果需要修改文件权限、所有者或用户组，需要使用 `RUN` 指令。另外，需要特别注意的是，如果我们给出的 `<src>` 是一个目录，目录本身是不会被复制进镜像的，被复制的是目录中的内容。

如果在 `<src>` 中给出网络地址，`Docker` 会有一些特殊的处理方式。比如在利用其他方式复制时，即使文件已经修改或者构建缓存被更新，文件的修改时间 (`mtime`) 是不会被带入的。如果在网络文件 HTTP 请求的响应头中出现 `Last-Modified`，`Docker` 会采用其给出的时间来设置被复制文件的修改时间。

**小提示：**如果 `ADD` 指令所给出的需要被复制的文件与构建缓存中的文件不符，也就是这个文件相对于上次构建后有修改，则 `ADD` 指令的执行就不会使用构建缓存，并且之后所有指令的执行，也不会再使用构建缓存。

虽然 ADD 指令能够获取到网络文件，但其本身并没有过于强大的网络访问功能，带有认证等机制的网络服务器，是不能通过 ADD 指令来获取其内容的。这时我们可以使用 RUN 指令去执行 wget、curl 或者其他专业的网络工具，来获取我们想要的内容。

除了能够下载网络内容，ADD 指令还能够自动完成对压缩文件的解压。如果我们提供的<src>是 Docker 能够识别的压缩文件格式（gzip、bzip2、xz），则其中的内容会解压到<dest>中，而非直接进行复制。Docker 识别文件是否是压缩文件，是根据文件内容中的特征码来实现的，如果一个文件并非压缩文件，而只是文件后缀为压缩文件的扩展名，那么 Docker 并不会尝试解压它。另外，自动解压文件只对本地文件有效，如果<src>给出的是网络路径，Docker 不会对文件进行解压。

## 5.4.2 COPY

在 Dockerfile 中还有一种引入文件的方式：使用 COPY 指令。与 ADD 指令非常相似，COPY 指令也有以下两种使用格式。

```
COPY <src>... <dest>
```

或：

```
COPY ["<src>",... "<dest>"]
```

COPY 指令在源路径<src>、目标路径<dest>，以及文件通配符的使用上，与 ADD 指令的规则几乎是一致的。主要的区别就在于 COPY 指令不能识别网络地址，也不会自动对压缩文件进行解压。在不需要自动解压或者没有网络文件需求的时候，使用 COPY 指令是一个不错的选择。

## 5.5 执行指令

执行指令能够指定通过镜像建立容器时，容器默认执行的命令。我们通常使用这些命令来启动镜像中的主要程序。

### 5.5.1 CMD

Docker 容器是为运行单独应用程序而设计的，当 Docker 容器启动时，实际上是对程序的启动，而容器是否停止也以程序是否结束为标准。而在 Dockerfile 中，就可以通



过 CMD 指令来指定由镜像创建的容器中的主体程序。

CMD 指令有以下三种使用格式。

```
CMD ["executable","param1","param2", ...]
```

或:

```
CMD ["param1","param2", ...]
```

或:

```
CMD command param1 param2 ...
```

其中, CMD command param1 param2 ...和 CMD ["executable","param1","param2",...] 的用法和 RUN 指令的使用方法是类似的,都是取决于是否使用 Shell 程序来执行命令。而 CMD ["param1","param2",...]这种格式则是将给出的参数传给 ENTRYPOINT 指令给出的程序。

需要注意的是,因为容器中只会绑定一个应用程序,所以在 Dockerfile 中只存在一个 CMD 指令,如果我们给出了多个 CMD 指令,之后的指令会覆盖掉之前的指令。

**提示:** 不要混淆了 CMD 指令和 RUN 指令,它们的区别是很大的。RUN 指令是在镜像构建的过程中执行,并将执行结果提交到新的镜像层中;CMD 指令在镜像构建的过程中不能执行,它只是配置镜像的默认入口程序。

对于 RUN 指令来说,我们更倾向于使用 RUN command param1 param2 ...这种格式,因为这种格式能够提供长指令换行的支持。而对于 CMD 指令来说, CMD ["executable","param1","param2",...]这种格式则相对较好,因为在这种格式下我们执行的是程序本身,所以容器运行绑定的也是程序本身,而非 Shell 程序。

虽然 CMD 指令指定了基于这个镜像所创建的容器在默认情况下运行的程序,但这个程序,准确地说是启动这个程序的命令,并非是不变的。我们在创建容器时,可以重新指定应用程序的启动指令,也就是说,我们可以修改需要运行的应用程序,而容器中给出的 CMD 指令会被覆盖掉。

```
$ sudo docker run -it nginx /bin/bash
```

如果我们通过上面这条指令创建和运行 Nginx 容器,容器启动后执行的是/bin/bash,也就是一个 Shell 程序,而我们也通过-it 指令绑定到了/bin/bash 程序上,可以在容器中进行操作,而 nginx 程序则并没有运行。也就是说,在这种使用方法下,容器并没有提供 Nginx 服务,而容器的停止也改为以/bin/bash 程序的结束为标准。

## 5.5.2 ENTRYPOINT

镜像所指定的应用程序在容器中运行时，难免需要一些系统服务或其他程序的支持和配合，我们可以在 CMD 指令中启动这些服务，但这样做会让启动服务的命令与启动主程序的命令杂糅在一起，显得比较混乱。ENTRYPOINT 指令就是专门用于主程序启动前的准备工作的。

使用 ENTRYPOINT 指令的方式与使用 CMD 指令的方式非常相似。

```
ENTRYPOINT ["executable", "param1", "param2", ...]
```

或：

```
ENTRYPOINT command param1 param2 ...
```

这两种格式在效果上与 CMD 指令是一样的，ENTRYPOINT command param1 param2...这种格式会使用 Shell 程序来执行，而 ENTRYPOINT ["executable", "param1", "param2", ...]则是直接执行程序。

需要特别注意的是，当 ENTRYPOINT 指令被指定时，所有的 CMD 指令或通过 docker run 等方式指定的应用程序启动命令，不会在容器启动时被直接执行，而是把这些命令当作参数，拼接到 ENTRYPOINT 指令给出的命令之后，传入 ENTRYPOINT 指令给出的程序中。由于这个特殊性，我们在使用 ENTRYPOINT 时需要特别注意使用的方式方法。

我们应该避免在使用 ENTRYPOINT 时把 CMD 指令的形式配置成 shell 格式，即 CMD command param1 param2 ...。因为如果这样做，在 ENTRYPOINT 里是以次级命令的方式启动 CMD 的 Shell 进程的，所以这个进程在容器中的 PID 并不是 1，Docker 就不会把容器的生命周期绑定到这个进程上。这就会使我们在外部对容器发出的信号无法发送到这个进程上，包括容器停止时发出的信号，而没有收到信号的程序会被强制结束，这就会造成数据的损坏以及其他意想不到的结果。

下面使用 ENTRYPOINT 指令和 CMD 指令来编辑一个 Dockerfile 文件：

```
FROM debian:jessie
ENTRYPOINT ["top", "-b"]
CMD ["-c"]
```

使用 docker build 命令将 Dockerfile 构建成镜像，并运行基于它的容器：

```
$ sudo docker build -t entry_test
Sending build context to Docker daemon 38.4 kB
Step 1 : FROM debian:jessie
```

```

---> 1b088884749b
Step 2 : ENTRYPOINT top -b
---> Running in 36656629b931
---> 5d7a4166a02b
Removing intermediate container 36656629b931
Step 3 : CMD -c
---> Running in 1b8e36cfc19
---> b0e20b0d2a47
Removing intermediate container 1b8e36cfc19
Successfully built b0e20b0d2a47
$ sudo docker run -it --rm entry_test
top - 04:53:59 up 8 days, 21:27, 0 users, load average: 0.22, 0.08, 0.06
Tasks:   1 total,      1 running,    0 sleeping,    0 stopped,    0 zombie
%Cpu(s):  0.1 us,      0.1 sy,       0.0 ni, 99.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.1 st
KiB Mem:  3619012 total, 3484272 used, 134740 free,   204816 buffers
KiB Swap:  0 total    0 used,       0 free. 1847464 cached Mem

PID USER   PR    NI  VIRT   RES    SHR  S   %CPU  %MEM   TIME+  COMMAND
  1  root    20     0   21804  1240   980  R    0.0   0.0   0:00.01 top -b -c

```

容器运行后，top 程序成了容器绑定的主进程，它也是这个容器中的唯一进程，进程的 PID 为 1。而在 top 返回的结果中，我们还能看到 top 本身的启动命令：top -b -c，结合我们在 Dockerfile 中的定义，很容易发现，Docker 在运行容器时是把 ENTRYPOINT 指令和 CMD 指令给出的命令进行了拼接并执行。

**小提示：**在使用容器的创建命令时，要特别注意选项参数与镜像名称及容器启动指令的位置，例如 docker run <image> -d 中的 -d，并不表示让容器进入后台运行，而是把 -d 当作参数传入容器的 ENTRYPOINT 给出的命令中。

要了解不同使用形式的 CMD 和 ENTRYPOINT 指令组合使用时的关系，可以参考表 5-1。

表 5-1 CMD与ENTRYPOINT指令组合执行时的关系

ENTRYPOINT指令		CMD指令
未定义		未定义
实际执行	错误指令，不能执行	
未定义		CMD ["cmd_exec", "cmd_p1"]
实际执行	cmd_exec cmd_p1	



续表

未定义	CMD ["cmd_p1", "cmd_p2"]
实际执行	cmd_p1 cmd_p2
未定义	CMD cmd_exec cmd_p1
实际执行	/bin/sh -c cmd_exec cmd_p1
ENTRYPOINT ["entry_exec", "entry_p1"]	未定义
实际执行	entry_exec entry_p1
ENTRYPOINT ["entry_exec", "entry_p1"]	CMD ["cmd_exec", "cmd_p1"]
实际执行	entry_exec entry_p1 cmd_exec cmd_p1
ENTRYPOINT ["entry_exec", "entry_p1"]	CMD ["cmd_p1", "cmd_p2"]
实际执行	entry_exec entry_p1 cmd_p1 cmd_p2
ENTRYPOINT ["entry_exec", "entry_p1"]	CMD cmd_exec cmd_p1
实际执行	entry_exec entry_p1 /bin/sh -c cmd_exec cmd_p1
ENTRYPOINT entry_exec entry_p1	未定义
实际执行	/bin/sh -c entry_exec entry_p1
ENTRYPOINT entry_exec entry_p1	CMD ["cmd_exec", "cmd_p1"]
实际执行	/bin/sh -c entry_exec entry_p1 cmd_exec cmd_p1
ENTRYPOINT entry_exec entry_p1	CMD ["cmd_p1", "cmd_p2"]
实际执行	/bin/sh -c entry_exec entry_p1 cmd_p1 cmd_p2
ENTRYPOINT entry_exec entry_p1	CMD cmd_exec cmd_p1
实际执行	/bin/sh -c entry_exec entry_p1 /bin/sh -c cmd_exec cmd_p1

在大多数情况下，我们更希望 ENTRYPOINT 指令与 CMD 指令是分离的，CMD 指令可以单独运行，有利于我们在镜像中指定其他程序作为主程序。另外，在 ENTRYPOINT 指令中，我们通常也不会只执行一条语句，而是需要较多的命令为应用程序准备运行依赖，而这对于只能书写简单指令的 ENTRYPOINT 显然不适用。此时，我们可以使用 Shell 执行脚本来帮助我们达到这两个目的。

我们定义一个启动脚本：

```
#!/bin/bash

# 执行准备和启动依赖程序
...

# 执行主程序
exec "$@"
```

在这个脚本中，我们可以加入需要的基本环节准备工作任务，也可以放置启动依赖程序的命令，在脚本的最后，我们通过 `exec` 命令启动主程序。这里我们用到了 `exec` 命令的特性，即脚本会直接衔接到主程序上，这样就能让我们定义的主程序正常地收到传递给容器的信号了。

我们可以将脚本程序保存为 `start.sh`，并放置在 `Dockerfile` 相同的目录下，然后可以如下所示定义 `Dockerfile` 文件：

```
...
# 复制脚本到镜像中
COPY start.sh /start.sh

# 给脚本添加执行权限
RUN chmod +x /start.sh

# 定义 ENTRYPOINT
ENTRYPOINT ["/start.sh"]
...
```

因为我们将 `start.sh` 复制到镜像中时，脚本文件默认是没有执行权限的，所以不要忘记增加脚本文件的执行权限。

与 `CMD` 指令一样，`ENTRYPOINT` 指令也并非镜像指定后就不能被修改，我们创建基于这个镜像的容器时，可以通过 `--entrypoint` 参数来覆盖镜像中原有的 `ENTRYPOINT` 指令。需要注意的是，我们只能使用可执行程序来替换 `ENTRYPOINT` 指令，因为被 `--entrypoint` 替换的命令是直接执行的，而不是通过 `Shell` 程序执行的。

## 5.6 配置指令

若想对镜像或者通过镜像所创建的容器进行相关环境或者网络、用户等的配置，可以通过配置指令来实现。

## 5.6.1 EXPOSE

每个容器都有自己的端口系统，相互之间不连通也不共享，容器间的网络通信是需要通过 Docker 转接来完成的。如果容器中的应用程序需要让其他镜像访问到它提供的端口，需要显式给出对外提供的端口号。而没有给出的端口号，即使容器中的应用程序对它们进行了监听，但 Docker 没有对它们进行转接，所以在容器外依然是访问不到的。

**提示：**在 Docker 新的网络模块特性中，已经允许创建网络时就不再需要主动敞口了。

要指定基于我们将要生成镜像的容器对外敞开的端口，可以使用 EXPOSE 指令。EXPOSE 指令的使用方法很简单，将需要共享的端口逐一传入即可：

```
EXPOSE <port> [<port>...]
```

需要注意的是，EXPOSE 指令所指定的端口与 docker run 等命令中 -P 或 -p 参数所指定的端口，在用法和含义上是有很区别的。EXPOSE 指令是给出基于此镜像的容器需要敞开的端口，这些端口可以从容器外部访问到。而创建容器时所使用的 -P 或 -p 参数，都是用于建立容器到宿主机外部端口的映射的。

也就是说，要从外部访问容器内程序监听的端口，首先需要通过 EXPOSE 指令将这些端口标记为对外敞开，再根据实际访问的来源进行配置：从其他容器中访问则需要创建该容器时使用 --link 连接到此容器；从宿主机外访问则需要创建该容器时使用 -p 或 -P 参数建立宿主机对外端口与容器端口的转发。

## 5.6.2 ENV

在 Dockerfile 中，我们也能指定环境变量，环境变量能够替换 Dockerfile 中其他指令出现的参数，使用 ENV 指令就能很轻松地设置 Dockerfile 中的环境变量。

```
ENV <key> <value>
```

或：

```
ENV <key>=<value> ...
```

使用 ENV <key> <value> 这种格式可以指定一个环境变量，在键名之后的数据都会被视作环境变量的值，所以值中并不需要转义符号：

```
ENV myDog The Dog
ENV myCat The Cat
ENV myFish The Fish
```



而 `ENV <key>=<value> ...` 这种格式能够一次指定多个环境变量,并且可以使用 `\` 进行换行连接,但与此同时,参数的值是需要进行转义的:

```
ENV myDog="The Dog" myCat=The\ Cat \
myFish=The\ Fish
```

这两种书写形式所达到的效果是一致的,不过我们依旧推荐使用第二种写法,因为多次使用 `ENV` 指令也会生成多个进行层。

环境变量也是能够继承的,所有在基础镜像中设置的环境变量,也都会被继承到将要构建的镜像中。而且,在 `Dockerfile` 中所指定的环境变量,不但会影响镜像构建过程中的指令,也会在基于此镜像的容器运行时产生效果。我们可以通过 `docker inspect` 命令查看镜像中的环境变量,在容器创建时也可以通过 `--env` 参数来新增和替换环境变量:

```
$ sudo docker run --env <key>=<value> ...
```

因为环境变量是持久存在的,所以可能会造成一些意想不到的结果。例如,如果给出一条环境变量 `ENV DEBIAN_FRONTEND noninteractiv`,这在基于 Debian 系统镜像的 `Dockerfile` 中,会对 `apt-get` 的使用产生影响。如果只想在某一条指令中使用环境变量,可以使用这种形式的 `RUN` 指令:

```
RUN <key>=<value> command param1 param2 ...
```

### 5.6.3 LABEL

使用 `LABEL` 指令,可以为即将生成的镜像提供一些元数据作为标记,这些标记能够帮助我们更好地展示镜像的信息。

`LABEL` 指令的用法很简单:

```
LABEL <key>=<value> <key>=<value> <key>=<value> ...
```

在 `LABEL` 指令之后,带入我们希望加入的元数据的键值对,如果有多个键值对,可以使用空格分隔它们。在键和值中,如果带有空格,可以使用引号包裹,避免与分隔符产生歧义。而如果数据过长,也可通过 `\` 进行换行。

```
LABEL "com.example.vendor"="You Ming"
LABEL com.example.label-with-value="foo"
LABEL version="1.0"
LABEL description="This text illustrates \
that label-values can span multiple lines."
```

我们可以在 Dockerfile 中连续使用多个 LABEL 指令来指定不同的标记,不过更推荐连接所有标记到一个 LABEL 指令中去,因为每个 LABEL 指令都会产生一个新的镜像层,如果我们给出了较多的标记并且分别使用 LABEL 指令,就会造成镜像层的浪费。

在连接标记时,还可以加入换行符,如:

```
LABEL multi.label1="value1" multi.label2="value2" other="value3"
```

可以写成:

```
LABEL multi.label1="value1" \  
      multi.label2="value2" \  
      other="value3"
```

镜像的元数据是存在继承关系的,从 FROM 指令给出的基础镜像中所有的标签也会被加入到新的镜像中。而如果我们使用 LABEL 指令给出的标签中含有重名的标签,新的数据会覆盖原有的数据。

可以通过 docker inspect 命令查看镜像的标记:

```
...  
"Labels": {  
  "com.example.vendor": "You Ming"  
  "com.example.label-with-value": "foo",  
  "version": "1.0",  
  "description": "This text illustrates that label-values can span multiple lines.",  
  "multi.label1": "value1",  
  "multi.label2": "value2",  
  "other": "value3"  
},  
...
```

## 5.6.4 USER

USER 指令用于设置执行用户,我们可以传入用户的名称或 UID 作为 USER 指令的参数。

```
USER nginx
```

需要注意的是,USER 指令对其后的 RUN 指令、CMD 指令和 ENTRYPOINT 指令都会起作用。也就是说,如果使用 USER 指令,不但会影响基于此镜像的容器中主程序的运行用户,也会影响在 USER 指令之后在构建过程中通过 RUN 指令给出的命令。

## 5.6.5 ARG

在构建过程中，我们时常还需要进行配置或使用一些变量，ARG 指令就为我们提供了设置变量的方法。

ARG 指令与 ENV 指令有很大的不同。ENV 指令用于配置环境变量，它的值会影响镜像的编译，也会体现在容器的运行中，需要改变环境变量时，要在容器启动时进行赋值。而 ARG 指令则只用于镜像的构建过程中，其效果不会作用于基于此镜像的容器，而覆盖参数的方式也是通过 docker build 中的 --build-arg 来进行的。

使用 ARG 指令定义参数的方式很简单。

```
ARG <name>
```

或：

```
ARG <name>=<default>
```

我们可以注意到，在使用 ARG <name> 的形式定义变量时，并没有给出变量的值，因为我们定义的就是变量，而变量应该是由外部传递进来的，而非我们在 Dockerfile 中写出来的，所以这里只是进行了变量的声明。对于 ARG <name>=<default> 这种形式来说，“=” 之后的值，也只是在我们未提供变量时所使用的默认值。

下面展示一个基于 BusyBox 创建的容器，利用参数来指定它的运行用户：

```
FROM busybox

ARG user

USER $user

...
```

这里没有在 Dockerfile 中给出运行用户，因为我们不想过早地给出具体的用户。在真正构建镜像时，可以使用 --build-arg 对这个参数赋值：

```
$ sudo docker build --build-arg user=root ./busybox
```

**注意：**千万不要使用 ARG 参数配置一些登录密码、认证密钥之类的参数。因为镜像使用的构建参数，可以被任何能接触到镜像的人通过 docker history 命令查询到。

因为 ENV 指令和 ARG 指令所定义的参数，在使用时都是通过 “\$+参数名” 的形式来访问的，所以就可能造成变量定义冲突。在这种情况下，我们只需要记住，ENV 指令所定义的变量永远都会覆盖 ARG 指令所定义的变量即可，即使它们定义的顺序是相反的。



```
FROM ubuntu

ARG DEMO_VER

ENV DEMO_VER v0.1

RUN echo $DEMO_VER

...
```

在上例中，不论我们使用 `docker build` 的 `--build-arg` 为 `DEMO_VER` 赋何值，镜像里的 `DEMO_VER` 都会被环境变量覆盖为 `v0.1`。

我们可以通过这个技巧让环境变量也通过 `--build-arg` 来配置，只需要把 `Dockerfile` 定义为如下的方式即可：

```
FROM Ubuntu

ARG DEMO_VER

ENV DEMO_VER $DEMO_VER

RUN echo $DEMO_VER

...
```

这样就可以使用 `--build-arg` 将 `DEMO_VER` 传递到构建参数中，而参数又会赋值给环境变量，实现了动态配置环境变量的目的。

Docker 为所有镜像预置了几个变量，我们可以直接在 `Dockerfile` 中使用它们，不再需要通过 `ARG` 指令来声明，而修改它们的方法仍然是在构建时使用 `--build-arg` 参数。这些被预置的参数是：`HTTP_PROXY`、`http_proxy`、`HTTPS_PROXY`、`https_proxy`、`FTP_PROXY`、`ftp_proxy`、`NO_PROXY`、`no_proxy`。

## 5.6.6 STOPSIGNAL

当我们停止容器时，Docker 会向容器中的应用程序传递停止信号，我们可以通过 `STOPSIGNAL` 指令来修改 Docker 所传递的信号。

`STOPSIGNAL` 支持两种格式定义的信号，一种是用 Linux 内核 `syscall` 信号的数字表示，另一种是用信号的名称表示。

```
STOPSIGNAL 9
```

或：

```
STOPSIGNAL SIGKILL
```

## 5.6.7 SHELL

之前我们说到过, CMD、ENTRYPOINT 等指令都支持以 shell 形式执行, 而 SHELL 指令可以为它们选定 Shell 程序。SHELL 指令的使用格式如下:

```
SHELL ["executable", "parameters"]
```

Docker 默认使用的 Shell 程序是 /bin/sh, 若要改为 /bin/bash, 可以使用如下 SHELL 指令:

```
SHELL ["/bin/bash", "-c"]
```

## 5.7 特殊用法

除了基本的指令及备注信息, 在 Dockerfile 中, 我们还能通过一些特殊的使用方法控制镜像的构建过程, 以及关于 Dockerfile 解析中的处理方式。

### 5.7.1 环境变量

通过 ENV 指令定义环境变量后, 就可以在之后的命令中进行环境变量的替换了。环境变量的解析支持 ADD、COPY、ENV、EXPOSE、LABEL、USER、WORKDIR、VOLUME、STOPSIGNAL 这几个指令。当然, 如果我们在 ONBUILD 中使用了上述指令, 那么它们也可以在 ONBUILD 里被解析。

普通的环境变量替换方法是使用 “\$+变量名” 的方式:

```
ENV variable value
RUN echo $variable # value
```

也可以使用花括号将变量名包裹起来, 以避免出现歧义:

```
ENV variable value
RUN echo $variable # value
RUN echo ${variable}_1 # value_1
```

如果使用变量的方法刚好是我们想要使用的内容, 可以使用转义符号去除环境变量的解析过程:

```
ENV variable value
RUN echo \$variable # $variable
```

使用环境变量时，我们还可以进行简单的判断。例如，使用 `${variable:-word}` 表示当环境变量 `variable` 不存在时，使用 `word` 进行替换。使用 `${variable:+word}` 表示如果环境变量 `variable` 已经被定义和赋值，`word` 会替换占位符；如果环境变量 `variable` 没有被设置，占位符会被直接清除。

## 5.7.2 指令解析

之前我们讲解过，使用 `RUN` 等指令时，可以通过 `\` 来进行命令的换行。这对于 `Linux` 或 `Mac OS` 系统是比较适用的，但 `Windows` 系统本身的目录分隔符就是 `\`，这样书写就会造成歧义。要解决这个问题，就需要利用 `Dockerfile` 中注释的一种特殊用法：解析指令行（`Parser Directives`）。

解析指令行的一般用法是：

```
# directive = value
```

参数名和值分布在等号的两端，参数名是区分大小写的，并且参数名与值周围的空格也会被忽略掉，所以下列几种用法的效果是一致的：

```
#directive=value
# directive =value
# directive= value
# directive = value
# dIrEcTiVe=value
```

使用 `escape` 这个解析参数，可以设置换行分隔符，用法也很简单：

```
# escape=
```

我们可以对比一下没有使用正确分隔符和使用了正确分隔符的 `Dockerfile` 在构建后会出现的结果。

下面写一个简单的 `Dockerfile`，加上一些在 `Windows` 系统中路径的使用，并且不带有 `escape` 解析指令参数：

```
FROM windowsservercore
COPY testfile.txt c:\\
RUN dir c:\\
```

之后再尝试对 `Dockerfile` 进行构建。

```
Sending build context to Docker daemon 3.072 kB
Step 1 : FROM windowsservercore
```



```

---> dbfee88ee9fd
Step 2 : COPY testfile.txt c:RUN dir c:
GetFileAttributesEx c:RUN: The system cannot find the file specified.

```

此构建过程会以失败告终，因为 RUN 指令没有找到对应的路径，路径分隔符被错误地解析并删除了。

接下来使用一个携带 escape 解析参数的 Dockerfile:

```

# escape=
FROM windowsservercore
COPY testfile.txt c:\
RUN dir c:\
Results in:

```

尝试对这个 Dockerfile 镜像进行构建。

```

Sending build context to Docker daemon 3.072 kB
Step 1 : FROM windowsservercore
---> dbfee88ee9fd
Step 2 : COPY testfile.txt c:\
---> 99ceb62e90df
Removing intermediate container 62afbe726221
Step 3 : RUN dir c:\
---> Running in a5ff53ad6323
Volume in drive C has no label.
Volume Serial Number is 1440-27FA

Directory of c:\

03/25/2016  05:28 AM  <DIR>          inetpub
03/25/2016  04:22 AM  <DIR>          PerfLogs
04/22/2016  10:59 PM  <DIR>          Program Files
03/25/2016  04:22 AM  <DIR>          Program Files (x86)
04/18/2016  09:26 AM              4 testfile.txt
04/22/2016  10:59 PM  <DIR>          Users
04/22/2016  10:59 PM  <DIR>          Windows
                1 File(s)              4 bytes
                6 Dir(s)  21,252,689,920 bytes free
---> 2569aa19abef
Removing intermediate container a5ff53ad6323
Successfully built 2569aa19abef

```

构建的过程非常顺利，Windows 系统的路径分隔符都被正确地使用了。

### 5.7.3 忽略文件

在使用 docker build 进行镜像构建时，我们传递给 docker build 的目录会被当作构建的环境目录，其中的文件可以被 Dockerfile 所使用，并通过 COPY 或者 ADD 指令复制到镜像中。

某些时候，在给出的构建环境目录中，可能存在一些存储敏感信息，或者其他我们并不希望被构建所使用的文件或目录，可以通过类似 GIT 的忽略文件来忽略这些文件和目录。

忽略文件的名称应该是.dockerignore，是一个没有文件名只有扩展名的文件。在进行镜像的构建之前，Docker 先扫描给出的环境目录中是否存在.dockerignore 文件，如果文件存在，Docker 会读取其中的内容，并根据给出的规则对文件和目录进行忽略处理。

下面是一个简单的.dockerignore 文件：

```
# comment
temp
*/temp*
temp?
!temp.keep
```

与 GIT 忽略文件相似，Docker 的忽略文件也支持使用\*进行多字符匹配；使用?进行单字符匹配；使用!进行强制保留等。所以在上面的忽略文件示例中，我们可以发现如表 5-2 所示的几项忽略规则。

表 5-2 .dockerignore规则解析

规 则	解 析
# comment	备注信息，不会被看成规则进行处理
temp	忽略所有名称为temp的文件或目录
*/temp*	可以匹配并忽略/filedir/temporary.tmp这样的文件，也可以忽略/filedir/temp这样的目录
temp?	可以忽略以tempa或tempb等作为名称的文件或目录
!temp.keep	不论忽略规则是什么，temp.keep这个文件会被保留

我们甚至可以忽略.dockerignore 本身，不过它仍然会被 Docker 读取，但我们不能通过 COPY 或 ADD 指令将其复制到镜像中。

不同于 GIT 的忽略文件，在 Docker 中，我们更倾向于忽略掉所有的文件，只保留已知的需要传入镜像的文件。

```
# Keep files
*
!keepA
!keepB
...
```

## 5.8 本章小结

在本章中，我们了解和学习了使用 Dockerfile 去构建 Docker 镜像的过程，解析了每个 Dockerfile 指令的用法。Dockerfile 是 Docker 快速迁移镜像的秘诀，也是最常用的构建镜像的方式，不但使部署更便利，也是很多工具辅助管理镜像的首选配置方式。掌握 Dockerfile 的使用方法，是使用 Docker 镜像最关键的一环。

通过本章的例子，我们对构建镜像的过程有了简单的认识，在之后的章节中，我们将进行实践演练，并经常使用 Dockerfile 构建镜像。



## 第二部分 实践篇

# 第 6 章

## SSH 服务

对服务器进行操作，Shell 程序必然是输入指令和展示结果最有效的工具，每一个开发者或多或少都会对 Shell 程序有所了解，我们常说在 Linux 系统下输出某种命令，其实就是在 Shell 程序中输出和执行这个命令。对于运维人员来说，使用 Shell 程序更是家常便饭，几乎对服务器的所有操作，以及监测服务器的状态都要使用 Shell 程序。

如果不使用服务器，就需要使用互联网对服务器进行操作了，SSH (Secure Shell) 可以帮助我们在互联网中使用 Shell 的程序和协议。SSH 为我们在互联网中传递对服务器的操作，并对服务器返回的结果进行加密，确保远程操作服务器时的安全。

在本章中，我们就讲解 SSH 服务在 Docker 中的搭建过程，解读如何制作具备 SSH 服务的镜像。

### 6.1 在 Docker 中使用 SSH

SSH 服务作为远程操作服务主机的主要方式之一，在很多场景下都会使用到。在 Docker 中，我们也可以通过 SSH 连接和访问到容器的内部。

## 6.1.1 SSH 简介

作为远程操作服务器的利器，SSH 几乎成为每一台 Linux 服务器的标配。SSH 本身是一套定义在应用层和传输层上的安全协议，为计算机之间的登录以及 Shell 程序的调用提供安全的传输与使用环境。与其他传统的网络协议相比，SSH 的安全性更高。FTP、Telnet 等协议都使用明文在网络中传送账号、密码，甚至数据，而 SSH 则对在网络中传递的信息进行了加密，有效地防止了信息被泄露。

SSH 的优势在于，它留给了用户充分发挥的空间，如自定义算法、自定义密钥规则、扩展功能性协议等。比如我们可以使用 SSH 包裹 FTP 协议，让 FTP 协议在传输文件时变得更加安全。也有 GIT 一类的软件，支持使用 SSH 形式在客户端与服务器之间传递数据。

如果 SSH 只是一个协议，那么是怎么通过它连接服务器的呢？协议之于编程而言，更像一个未被实现的接口，我们要使用它，还需要寻找一款实现 SSH 协议的程序。有许多软件都实现了基于 SSH 协议的通信，如商业化的软件，也不乏开源的软件，在 Linux 平台下使用较多的应该是 OpenSSH，如图 6-1 所示。OpenSSH 不但实现了 SSH 的服务端和客户端，还提供了对 SFTP、密钥管理等功能的支持。



图 6-1 OpenSSH 的 Logo

## 6.1.2 SSH 使用方法简介

如果 SSH 要登录服务器，要先在服务器上启动 SSH 服务端程序。OpenSSH 会提供一个名为 SSHD 的程序用作 SSH 服务端程序。安装 OpenSSH 时，安装程序也会在系统服务中添加 SSHD 服务，可以通过 Linux 的服务控制命令启动 SSHD。

```
$ sudo service sshd start
```

系统服务都会以守护进程的方式启动，调用服务启动命令之后，其实 SSHD 程序并

不是在调用服务的进程中运行的。虽然在普通的 Linux 服务器系统中习惯以系统服务的方式启动 Linux，但是在 Docker 环境下，需要让容器绑定到 sshd 的进程上，因此要直接使用 SSHD 程序来启动。

在默认情况下，SSHD 会被安装到/usr/sbin 下，并直接运行启动 SSH 服务端程序。另外，SSHD 程序本身的启动方式就是守护态的，因此需要通过-D 参数使它转换到前台运行。

```
$ sudo /usr/sbin/sshd -D
```

当服务端程序启动之后，程序会监听主机的 22 端口，接收和处理客户端的请求，这时我们就可以通过客户端访问服务器了。

OpenSSH 还为我们提供了 SSH 客户端程序，我们在 Linux 环境下可以直接使用这个程序进行登录和操作。在 Windows 环境下，更常见的 SSH 客户端程序是 PuTTY。总之，不论使用哪个工具，使用 SSH 协议与服务器通信，以及对服务器进行的操作，基本都是一致的。

下面以登录为例来讲解 Linux 下的 SSH 程序的使用方式。

假如要登录到远程主机 host 的 user 用户中，并执行操作，可以使用 SSH 程序并附带下面的参数：

```
$ ssh user@host
```

SSH 支持多种登录鉴权方式，当客户端向服务端发送登录请求时，服务端会将配置中允许的鉴权方式及鉴权顺序发送给客户端。SSH 客户端根据鉴权顺序，将准备好可用的认证信息逐次提交给服务器。如果认证成功，服务器与客户端之间的安全通信连接就会被建立。

也可以不指定需要登录的远程用户：

```
$ ssh host
```

没有指定需要登录的用户时，会将 SSH 程序的用户名作为登录到服务器的用户名。

通过 SSH 登录到远程服务器的指定用户后，SSHD 会连接服务器用户的 Shell 程序。用户的 Shell 程序是在/etc/passwd 中指定的，可以从中查看用户的 Shell 程序。

```
$ sudo cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
```



```

sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
...

```

每行的最后记录的都是用户的 Shell 程序路径。我们可以看到很多用户指定的 Shell 程序是 `/usr/sbin/nologin`，这是一个不会提供 Shell 功能的 Shell 程序。也就是说，它的功能就像它的名字一样，是用来禁止这个用户登录的。当然，也可以通过修改 `/etc/passwd` 中这些用户的默认 Shell 程序让它们变得可以登录。

SSH 服务器程序默认监听的是主机的 22 端口，有时为了安全，会通过配置文件指定使用其他的端口。如果修改了端口，在客户端连接时会因为仍然采用默认的 22 端口进行连接而失败。此时可以在使用 SSH 程序时，带入 `-p` 参数来指定需要连接的 SSH 服务端程序所监听的端口，例如修改 SSH 服务端程序让其监听 2222 端口的时候，可以使用以下命令来连接：

```
$ ssh -p 2222 user@host
```

### 6.1.3 数据卷管理容器

根据 Docker 的特性，推荐将项目中不同的组件程序拆分到不同的容器中，形成在不同容器中的微服务。在开发、部署和维护这些应用程序以及包裹它们的容器的过程中，常常要对一些动态代码进行更新，对程序的配置进行更改，对输出的日志进行导出。

要实现这个过程，一个简单的方法就是利用数据卷将代码、配置及日志等目录，从宿主机中挂载到容器里，让容器直接操作宿主机的文件。但是这种方法让使用 Docker 封装的统一性不复存在，破坏了 Docker 所建立的安全结构，给日后维护带来了麻烦，也带来了不小的安全风险。另外，如果将代码、配置等部署在容器外，那么这些数据就不再享有 Docker 所带来的快速迁移的便利，也会给我们的工作带来不少麻烦。

从外界挂载数据卷到容器中并不是最佳的选择，我们可以尝试在数据卷容器中进行数据卷的挂载。数据卷容器很好地封装了数据卷及其中的数据，为数据的迁移提供了良好的适应性，也为整个项目中对 Docker 的使用提供了一致性保障。

虽然可以使用数据卷容器，但其也有弊端，我们并不能总是很方便地接触到数据卷容器中的数据。也就是说，直接修改存储在数据卷容器中的数据，是一个比较麻烦的过程。

不过这个缺陷是很好弥补的，如图 6-2 所示，可以将数据卷容器中的数据卷，挂载到另外一个独立容器中，再通过这个容器修改数据卷中的内容。因为通常情况下，都是通过网络访问服务器，所以可以直接通过 SSH 或 FTP 等能够进行网络文件传输的容器，来修改数据卷中的数据。

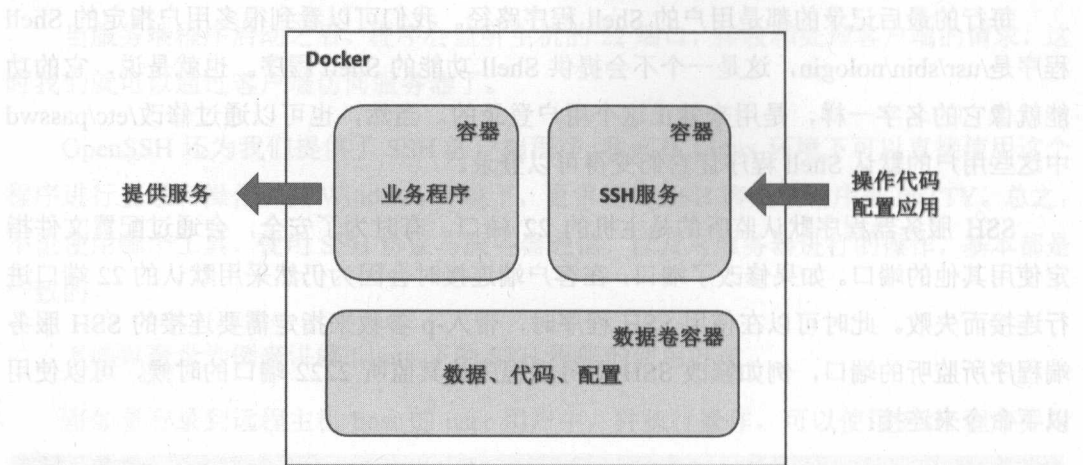


图 6-2 利用 SSH 服务管理代码与配置

使用 SSH 服务器管理数据，也能使安全性得到一定提升。即使 SSH 服务被攻破，攻击者能修改程序配置等信息，但是他们无法对运行在其他容器中的应用程序进行重启等操作，也就没有办法让修改的内容生效。另外，攻击者没有访问宿主机的权限，也就不能对服务器本身的配置产生影响。

## 6.1.4 使用 SSH 服务容器

当我们了解了使用 SSH 服务器来管理数据卷中的数据这种方式之后，再看看如何使用含有 SSH 服务的容器。

假设一个使用场景：若对外提供一个基于 Nginx 的 Web 服务，先将 Nginx 的配置和静态 HTML 文件代码分别放置在两个数据卷容器所提供的数据卷中。再创建一个能够提供 SSH 服务的容器，用来管理两个数据卷容器所提供的数据。

首先，分别创建存放配置和代码的数据卷容器，以及它们提供的数据卷。

```
$ sudo docker create --name config -v /etc/nginx alpine echo Nginx Configuration
$ sudo docker create --name code -v /var/web alpine echo Web Root
```

将数据卷容器分别命名为 **config**、**code**，可以帮助我们更好地区分不同容器的功能，也能有效避免混淆这两个容器。

再创建一个 **Nginx** 容器，分别从 **config** 数据卷容器和 **code** 数据卷容器中挂载配置和代码，并对外提供 **Web** 服务。

```
$ sudo docker create --name web --volumes-from code --volumes-from config -p 80:80
-p 443:443 nginx:1.10
```

创建好提供 **Web** 服务的容器之后，先不启动它。因为我们是从数据卷容器中挂载配置的，而现在存储配置的数据卷中还没有加入配置，所以还不能正确地启动 **Web** 服务。也就是说，应该先创建 **SSH** 容器，并通过 **SSH** 服务将配置加到存储配置的数据卷中。

在 **Docker Hub** 上有许多开发者贡献的能提供 **SSH** 服务的镜像，我们选择其中一个进行演示。

```
$ sudo docker pull rastasheep/ubuntu-sshd
Using default tag: latest
latest: Pulling from rastasheep/ubuntu-sshd

064f9af02539: Pull complete
390957b2f4f0: Pull complete
cee0974db2b8: Pull complete
c8144262002c: Pull complete
559e50b3108c: Pull complete
b591c117fbd5: Pull complete
c28d60bff361: Pull complete
f3f9100dac03: Pull complete
2d8700bbcc5f: Pull complete
16b3909f46fe: Pull complete
Digest: sha256:df90ab1d6a721b4a390cccd9460d69931eaalfble2ca790b5089c2e1aed30d60
Status: Downloaded newer image for rastasheep/ubuntu-sshd:latest
```

然后启动 **SSH** 容器，并挂载两个数据卷容器中的数据卷。

```
$ sudo docker run -d -P --name sshd --volumes-from code --volumes-from config
rastasheep/ubuntu-sshd
```

为了安全，这里使用 **-P** 参数让 **Docker** 随机分配外部地址给 **SSH** 容器的 22 端口。因为端口进行了随机分配，所以还需要查询 **Docker** 为我们分配的端口，以便进行 **SSH** 连接。



```
$ sudo docker port sshd 22
0.0.0.0:49187
```

之后就可以使用本地的 SSH 客户端工具，来连接这个由 SSH 容器提供的 SSH 服务器了。

```
$ ssh root@host -p 49187
```

这里所使用的 SSH 镜像默认开启了 root 这个用户的登录，登录密码为 root。可以在登录进入到 SSH 容器之后，修改用户密码，也可以新建其他用户来登录。

最后，通过 SSH 工具将配置文件和程序代码配置到对应的数据卷中。所有的准备工作完成之后，就可以启动 Web 服务了。

```
$ sudo docker start web
```

## 6.2 构建 SSH 服务镜像

我们可以利用 Docker 将 SSH 服务程序封装到 Docker 容器中，最佳的实践案例就是构建一个包含 SSH 服务的镜像。

### 6.2.1 构建方式比较

构建镜像的方式有两种。一种是新建容器，并在容器中按需求进行修改，再将容器提交为新的镜像。另一种是将构建镜像的指令全部写入到 Dockerfile 中，再通过 docker build 命令构建镜像。

我们更推荐使用 Dockerfile 来构建镜像。因为 Dockerfile 拥有更清晰的结构、更好的可移植性、更一致的构建结果。而且使用 docker commit 命令构建镜像时，无法保证构建镜像之前容器的状态是稳定的，也不能保证它是我们想要的。

但若我们对即将构建的镜像不完全了解时，建议先使用提交的方式进行尝试。因为通过提交的方式构建镜像，需要在容器中逐步完成镜像的搭建。在整个搭建过程中更易发现问题，并且可以立即处理。

若在不精确了解构建镜像的每个步骤的情况下，就直接使用 Dockerfile 构建镜像，难免会遇到依赖程序未安装、配置缺失、启动命令错误等问题。而解决这些问题需要修改 Dockerfile 并重新运行构建。这就会提高测试构建过程的成本，时间消耗也将更长。

所以，我们推荐在首次搭建新的应用程序镜像，或尝试更新应用程序镜像时，先打开容器，运行构建新镜像的命令，并检查运行的结果是否正确。如果在运行命令的过程中出现异常，则检查异常的原因，并解决问题。安装与配置完程序之后关闭容器，将容器提交为镜像，再通过新镜像运行新的容器，检查应用程序是否能正确运行起来。若能正确运行，再把这些准确的命令写入 Dockerfile 中，使用 Dockerfile 进行镜像分享、迁移等操作。

在之后的实践中，我们既会讲解通过逐步部署容器的方式构建镜像的过程，也会介绍构建镜像所使用的 Dockerfile。让大家在了解和使用不同软件构建镜像的过程中，掌握镜像是怎样产生出来的。熟悉了构建中的每个操作，不但能够丰富大家对搭建对应程序的了解，也能为解决程序运行中出现的问题，以及包裹这个程序的容器运行中出现的问题提供知识储备。

## 6.2.2 通过提交构建

下面使一个基础的操作系统镜像，建立、运行并进入容器，在容器的内部完成 OpenSSH 的安装与配置，并提交成 SSH 服务镜像。

这里使用 Ubuntu 16.04 版本的镜像作为 SSH 服务的基础镜像。先使用 /bin/bash 程序进入到镜像之中，准备进行 SSH 服务的搭建工作：

```
$ sudo docker run -it ubuntu:16.04 /bin/bash
root@blaae87c82aa:/#
```

因为 Ubuntu 系统是基于 Debian 的，所以 Debian Linux 系统中默认使用的程序包管理器 APT (Advanced Package Tool) 也被内置到 Ubuntu 之中。使用 APT 可以很轻松地在 Ubuntu 下安装软件，也可以自动解决软件的依赖问题。OpenSSH 程序能够在 APT 的软件仓库中找到，所以我们主要通过 APT 安装的方式，介绍搭建基于 OpenSSH 的 SSH 服务器的过程。

虽然 Ubuntu 的系统镜像会经常进行更新，但我们在使用时无法保障其中的软件处于最新的状态。所以需要先使用 APT 的更新命令对系统中已经安装的软件进行更新。

```
root@blaae87c82aa:/# apt-get update
Get:1 http://archive.ubuntu.com/ubuntu xenial InRelease [247 kB]
Get:2 http://archive.ubuntu.com/ubuntu xenial-updates InRelease [95.7 kB]
Get:3 http://archive.ubuntu.com/ubuntu xenial-security InRelease [94.5 kB]
Get:4 http://archive.ubuntu.com/ubuntu xenial/main Sources [1103 kB]
Get:5 http://archive.ubuntu.com/ubuntu xenial/restricted Sources [5179 B]
```

```
Get:6 http://archive.ubuntu.com/ubuntu xenial/universe Sources [9802 kB]
Get:7 http://archive.ubuntu.com/ubuntu xenial/main amd64 Packages [1558 kB]
Get:8 http://archive.ubuntu.com/ubuntu xenial/restricted amd64 Packages [14.1 kB]
Get:9 http://archive.ubuntu.com/ubuntu xenial/universe amd64 Packages [9827 kB]
Get:10 http://archive.ubuntu.com/ubuntu xenial-updates/main Sources [225 kB]
Get:11 http://archive.ubuntu.com/ubuntu xenial-updates/universe Sources [112 kB]
Get:12 http://archive.ubuntu.com/ubuntu xenial-updates/main amd64 Packages [473 kB]
Get:13 http://archive.ubuntu.com/ubuntu xenial-updates/universe amd64 Packages [398 kB]
Get:14 http://archive.ubuntu.com/ubuntu xenial-security/main Sources [42.1 kB]
Get:15 http://archive.ubuntu.com/ubuntu xenial-security/universe Sources [9622 B]
Get:16 http://archive.ubuntu.com/ubuntu xenial-security/main amd64 Packages [161 kB]
Get:17 http://archive.ubuntu.com/ubuntu xenial-security/universe amd64 Packages
[46.3 kB]
Fetched 24.2 MB in 6min 48s (59.3 kB/s)
Reading package lists... Done
```

软件更新完成之后,就可以进行 OpenSSH 的安装了。OpenSSH 的服务端程序在 APT 软件仓库中的名称为 `openssh-server`, 我们使用 `apt-get install` 命令来安装它。

```
apt-get install -y openssh-server
Reading package lists... Done
Building dependency tree
Reading state information... Done
...
```

因为我们要对安装命令进行总结,编写 Dockerfile。为了使 Dockerfile 编译时无须再进行确认,就能够完成自动化构建工作,所以在使用 APT 安装命令时,带入了 `-y` 参数。这个参数能够自动同意安装中所有的询问项目,帮助我们完成静默安装的过程。

当 OpenSSH 的 SSH 服务端程序安装完成后,不要急于退出并提交容器为镜像。因为我们需要通过用户来登录,要先创建用于登录的账号和设置登录密码。

```
root@b1aae87c82aa:/# groupadd wgroup
root@b1aae87c82aa:/# useradd -g wgroup wuser
root@b1aae87c82aa:/# passwd wuser
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
```

如果使用 `root` 用户登录,还要做特别的配置,因为默认情况下,OpenSSH 服务端程序禁止了 `root` 用户的密码登录。SSH 服务端程序的配置放置在 `/etc/ssh/sshd_config` 文件中,而 `PermitRootLogin` 的配置项表述的就是允许 `root` 用户登录的方式。



```
root@blaae87c82aa:/# more /etc/ssh/sshd_config | grep PermitRootLogin
PermitRootLogin prohibit-password
...
```

由上可以看到，在默认情况下，root 用户的登录配置是禁止密码登录的，只有将这项配置修改为 yes 才能打开 root 用户的密码登录。

由于包括 Ubuntu 在内的 Docker 系统镜像都比较干净，非必需的软件都没有内置到镜像之中，所以不能直接在镜像中找到 Vim 编辑器。当然，也可以通过 APT 来安装 Vim，但 Vim 不是 SSH 运行所必需的软件，所以为了避免浪费不必要的镜像层空间，一般不会对构建镜像的过程中安装无关软件。因此，这里就不使用 Vim 来编辑配置文件，而是通过 sed 编辑命令来修改配置。

```
root@blaae87c82aa:/# sed -i 's/PermitRootLogin prohibit-password/PermitRootLogin
yes/' /etc/ssh/sshd_config
root@blaae87c82aa:/# more /etc/ssh/sshd_config | grep PermitRootLogin
PermitRootLogin yes
...
```

Linux 中的 sed 命令可以对文件中的指定行进行操作，这里使用了 -i 参数来替换指定的内容。

上述工作都结束之后，就可以关闭容器并将对容器的更改提交为镜像了。

```
root@blaae87c82aa:/# exit
exit
$ sudo docker commit -m "SSH Server" blaae87c82aa ymdot/sshd
sha256:4ff364e083c17259e727ca0c46a52e7888177094924dc55e6956056c4b53a4d0
$ sudo docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
ymdot/sshd	latest	4ff364e083c1	44 hours ago	220.6 MB

```
...
```

至此，通过提交容器的方式构建运行 SSH 服务镜像的工作就完成了。

### 6.2.3 使用 Dockerfile 构建

下面开始了解如何使用 Dockerfile 来构建含有 SSH 服务的镜像。

我们依据之前在容器中搭建 SSH 服务的操作，可以很轻松地将整个过程写成一个 Dockerfile 文件。

```
# SSH Server
# VERSION 0.0.1
```

```
# 基础镜像
FROM ubuntu:16.04

# 维护者信息
MAINTAINER You Ming <youming@funcuter.org>

# 更新软件和安装 OpenSSH
RUN apt-get update && apt-get install -y openssh-server

# 建立 OpenSSH 的运行目录
RUN mkdir /var/run/sshd

# 在构建时设置 root 账户的密码
RUN echo 'root:hellossh' | chpasswd

# 打开 root 账户的密码登录
RUN sed -i 's/PermitRootLogin prohibit-password/PermitRootLogin yes/' /etc/
ssh/sshd_config

# 对外暴露 22 端口供 SSH 客户端连接
EXPOSE 22

# 启动命令，携带-D 参数使 SSHD 能在前台运行
CMD ["/usr/sbin/sshd", "-D"]
```

在 Dockerfile 中，使用了 `chpasswd` 来初始化 root 用户的密码。`chpasswd` 可以批量修改用户密码，只需要按照 `<user>:<pass>` 的形式传入需要修改的用户和密码即可。这里设置的初始密码为 `hellossh`，密码可以在我们建立容器并通过 SSH 客户端登录到容器中后，再通过 `passwd` 或 `chpasswd` 命令进行修改。

保存编写好的 Dockerfile，将 Dockerfile 所在的目录传入 `docker build`，再开始构建 SSH 镜像。

```
$ sudo docker build -t ymdot/sshd ./sshd
Sending build context to Docker daemon 2.56 kB
Step 1 : FROM ubuntu:16.04
----> f8d79ba03c00
Step 2 : MAINTAINER You Ming <youming@funcuter.org>
----> Running in 78beb9395f2c
----> ebb164d0bdd5
Removing intermediate container 78beb9395f2c
Step 3 : RUN apt-get update && apt-get install -y openssh-server
----> Running in c7872450bef0
...
Step 4 : RUN mkdir /var/run/sshd
```

```

---> Running in bc44465fc92d
---> 396366b41f27
Removing intermediate container bc44465fc92d
Step 5 : RUN echo 'root:hellossh' | chpasswd
---> Running in 8e8d3cd040bf
---> e29ece4ea693
Removing intermediate container 8e8d3cd040bf
Step 6 : RUN sed -i 's/PermitRootLogin prohibit-password/PermitRootLogin yes/'
/etc/ssh/sshd_config
---> Running in 6dca9537d351
---> efcde5e772ce
Removing intermediate container 6dca9537d351
Step 7 : EXPOSE 22
---> Running in e26bf3aa4746
---> 39d02e883912
Removing intermediate container e26bf3aa4746
Step 8 : CMD /usr/sbin/sshd -D
---> Running in 274321524b50
---> e8ba41d6a5da
Removing intermediate container 274321524b50
Successfully built e8ba41d6a5da

```

构建过程与在容器中搭建 OpenSSH 程序的过程一样，先通过 APT 更新基础镜像中的软件，再进行 OpenSSH 的安装。然后逐一完成建立运行目录、初始化 root 密码、运行 root 用户登录等工作。

在本地镜像列表中，可以找到刚刚构建的 SSH 服务镜像。

```

$ sudo docker images

```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
ymdot/ssh	latest	e8ba41d6a5da	14 minutes ago	220.3 MB
...				

我们通过这个镜像启动新的容器，来检查提供 SSH 服务的镜像，即刚刚构建的镜像是否能够正常运行。为了不占用宿主机 SSH 服务所使用的 22 端口，使用宿主机上另外一个端口来绑定 SSH 容器提供的 SSH 服务。

```

$ sudo docker run -d -P --name sshd ymdot/ssh
11ce44fb73069c0a0f3b503695c4457c2ff9a4554ac7795104f0c5ee3c709956
$ sudo docker ps -l

```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
11ce44fb7306	ymdot/ssh	"/usr/sbin/sshd -D"	19 minutes ago	



Up 19 minutes

0.0.0.0:32769-&gt;22/tcp sshd

容器运行起来后，我们尝试通过 SSH 客户端来登录它所提供的 SSH 服务。在 Windows 平台，可以使用 Xshell 一类的集成工具来实现 SSH 登录。在 Xshell 中输入完成连接信息，并单击“连接”按钮，在程序与运行在容器中的服务器之间建立连接之后，如图 6-3 所示，就能看到输入密码的提示框。

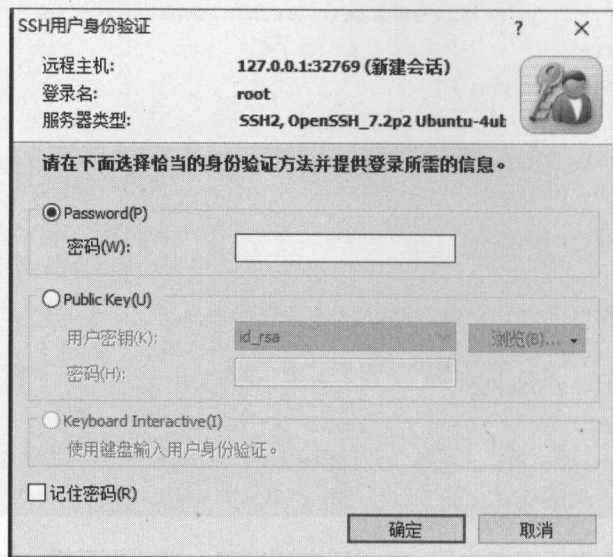


图 6-3 输入密码的提示框

输入完成之后会发现 Xshell 已经得到了容器中 SSH 服务的授权，并通过 SSH 协议进入到容器中的 Shell 环境下。容器中虚拟系统的信息也被 Shell 程序展示，并通过 SSH 服务端程序传送到了 Xshell 的界面中。

```
Connecting to 127.0.0.1:32769...
Connection established.
To escape to local shell, press 'Ctrl+Alt+]'.

Welcome to Ubuntu 16.04.1 LTS (GNU/Linux 4.4.15-moby x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.
```

```
Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.
```

```
root@11ce44fb7306:~#
```

实践证明，我们所构建的 SSH 镜像是可以正常使用的。利用 Docker 的特性，我们也可以将这个 Dockerfile 移动到其他宿主机中构建镜像。

## 6.3 本章小结

在本章中，我们介绍了通过使用 SSH 容器，在保证安全和封闭性的同时，维护整个 Docker 项目。我们还进行了构建 SSH 镜像的实践，对比了构建镜像的两种方式的优劣，并分别介绍了如何使用它们来构建 SSH 镜像。

通过本章的学习，大家不但了解了 SSH 服务在 Docker 中的作用，也掌握了在 Docker 中将独立应用程序封装成镜像的方法。

# 第 7 章

## Web 服务器

对于任何支持互联网连接的项目，特别是面向广大用户的项目来说，提供 Web 服务几乎属于基本配置。Web 服务因其简单性、丰富性，特别是界面对用户的友好性，一直以来就是互联网上传递信息最重要的方式之一。而 Web 服务器程序作为提供 Web 服务的主体，我们在开发和运维过程中自然会经常使用到。

在本章中，我们就对 Apache、Nginx、Tomcat 这几个 Web 服务程序在 Docker 中的搭建和运行方法进行讲解和实践。

### 7.1 Web 服务简介

#### 7.1.1 万维网与网站

得益于互联网连接全球的特性，我们能够与世界各地连接到互联网中的主机进行交流。为了让用户更方便地获取网络中其他主机中的信息，开发人员将这些信息都打包成资源，使用不同的资源标识符（URI）来访问，并通过超文本（Hypertext）等形式在网络中传播。这个由分布在全球的资源所组成的网络即万维网（World Wide Web）。

如图 7-1 所示，用户在浏览器中输入网址并按下回车键后，就可以寻找网址所对应资源所在的服务器。在告知拥有资源的服务器用户希望获得这个资源后，服务器为用户



准备好需要的资源，并通过超文本的方式将信息传递到电脑中，浏览器会根据资源类型选择合适的方式将信息展现给用户。

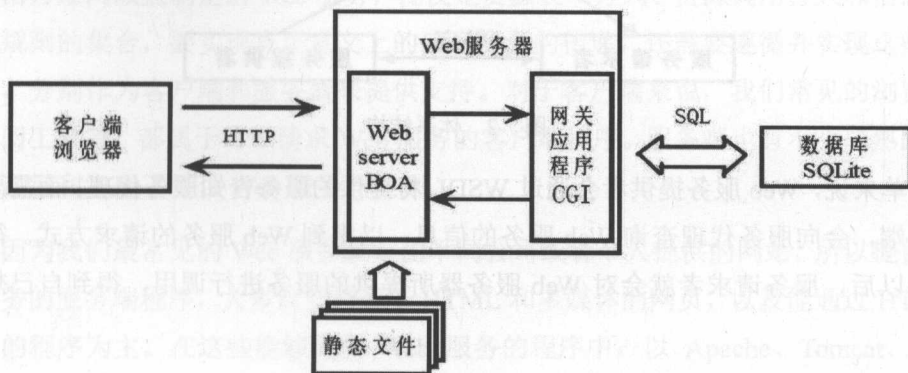


图 7-1 访问互联网的过程

为了让传递在万维网中的资源在传递形式上更统一，方便不同的主机、不同的程序、不同的用户获取万维网中的资源，万维网联盟于 1994 年在美国麻省理工大学成立。万维网联盟旨在为万维网指定一套统一的标准，我们耳熟能详的 HTML、CSS、XML 等都是由万维网联盟设计的，或者是得到万维网联盟认可的标准。

HTML 等标准是用户在浏览器中最常用的展示资源的方式。它通过标记语法，将不同的资源加以组合，并通过浏览器展示给用户，这就是我们常说的网页。而在网页中，又不乏指向其他资源的超链接（Hyperlink），这些链接所指向的由相同的提供者提供的网页所组成的集合，我们通常称为网站。

随着万维网联盟对规范的不扩充，特别是 HTML5 和 CSS3 标准被广泛使用后，用户可以从浏览器所展示的网页中获得越来越丰富的资源。而越来越多的企业，特别是期望使用互联网来推广、扩张服务及影响力的企业，开始使用万维网为用户提供服务。

## 7.1.2 Web 服务

如图 7-2 所示，万维网定义了以资源为核心的互联网信息传递方式，有了信息的传递方式，自然需要有能够以万维网的方式提供信息资源的程序，Web 服务就是这样的软件系统。虽然万维网联盟没有一个专门针对 Web 服务进行定义的标准，但是我们通常将遵循和实现了简单对象访问协议（SOAP），Web 服务描述语言（WSDL），统一描述、发现和集成（UDDI）这三者的程序，理解为 Web 服务程序。

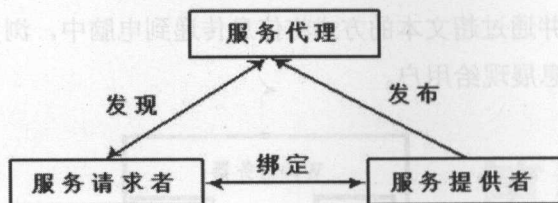


图 7-2 体系结构

简单来说，Web 服务提供者会通过 WSDL 将提供的服务告知服务代理。而需要服务的客户端，会向服务代理查询 Web 服务的信息，以得到 Web 服务的请求方式。得到调用方法以后，服务请求者就会对 Web 服务器所提供的服务进行调用，得到自己想要的信息。

Web 服务并不是单一的，它由多种工具组成，不同的工具调用的方法往往也不同。我们常见的远程过程调用（RPC）、服务导向架构（SOA）、表述性状态转移（REST）都是调用 Web 服务的方法之一。

早期 Web 服务的程序提供一个分布式函数或方法作为接口，让访问者进行调用。这种远程过程调用方式对服务端与客户端的一致性要求很高，因为客户端需要具体地知道 Web 服务程序提供了哪些接口，才能进行获取数据的调用过程。显然，这种高耦合的方式需要建立非常统一的标准，而在已经有很多独立规范的万维网中再制定一套统一标准是比较困难的。在一次次的批评中，许多 Web 服务提供商逐渐放弃了远程过程调用的方式，有的甚至已经在新的版本或标准中删除了远程过程调用的功能。

随着时代的发展，以资源状态为导向的表述性状态转移这种调用方法，越来越受到开发者的追捧。因为 REST 结构操作的主体就是万维网中最基础的信息单位——资源，而操作的方法也只有少数几个标准动作（HTTP 的 GET、PUT、DELETE 等），所以表述性状态转移的形式是一套非常简单的标准。因为它实现简单，所以任何 Web 服务程序都可以轻松地对它进行支持，它也刚好能弥补和解决远程过程调用中难以建立统一标准的问题。

虽然有多种提供给访问者的 Web 服务调用方式，但在传输协议方面，HTTP 已经成为 Web 服务传输方式的代名词。网页为用户提供了包罗万象的信息，而基于浏览器与 Web 服务的沟通几乎都是通过 HTTP 来进行的。由于受众广泛，其他协议很难取代 HTTP 的地位。

虽然 Web 服务提供的服务各具特色，使用的方式也各不相同，但是万变不离其宗，它们都是提供给客户端访问和获取服务端资源的方法或规则。

### 7.1.3 Web 服务程序

由万维网联盟制定的 Web 服务，仅仅是资源表示方式、资源调用方式和信息传递方式等规则的集合，要实现真正意义上的信息资源的传递，还需要遵循并实现这些规则的程序，分别作为客户端和服务端来提供支持。对于客户端来说，我们常见的浏览器、咨询订阅工具等，都属于可以请求 Web 服务的客户端程序。服务端也有不少成熟的应用程序，遵循和实现了提供 Web 服务的规则。

因为我们最常见的 Web 服务就是由不同公司或者个人提供的网站，所以提供这些网站服务的服务端程序，大多以支持基于 HTML 和多媒体的网页，以及能通过 HTTP 进行传输的程序为主。在这些能够提供 Web 服务的程序中，以 Apache、Tomcat、Nginx、WebLogic 比较常用，它们都提供了 HTTP 请求处理并返回基础的 HTML 和多媒体资源的功能，并且能支持对其他 Web 服务功能的延伸或扩展支持。

在常用的服务器架构中，通常采用这些 Web 服务器作为处理客户请求的第一道关卡，处理静态资源的调用、部分缓存规则、HTTP 约定等方面的事务。对于需要动态响应的内容，则由这些 Web 服务器向后端的动态处理程序调用，得到结果并输出给客户端来完成。

Web 服务程序是搭建 Web 服务器，特别是搭建网站不可或缺的程序之一。了解 Web 服务的概念和 Web 服务程序搭建的过程，也是开发者和运维人员需要掌握的基本技能之一。

## 7.2 Apache

Apache 是目前使用范围最广的 Web 服务程序，它有丰富的能够提供 Web 服务的功能。

### 7.2.1 Apache 简介

Apache HTTP Server 是一款开源的 Web 服务器程序，支持在多种操作系统下运行，具有跨平台性和安全性，被广泛地使用在很多网站的服务器中。

作为一款相对重量级的 Web 服务器，Apache 支持从基础验证到动态处理程序、从



TLS 安全层到代理服、从访问日志到请求过滤等功能。这些功能都以模块的形式出现在 Apache 中，开发者可以自己开发模块来扩展 Apache。可定制的模块让 Apache 拥有了强大的延展性，也让 Apache 在长期发展中积累了大量包含不同功能、能提供不同形式的支持的扩展模块。

Apache 由美国伊利诺伊大学香槟分校的国家超级电脑应用中心开发，在逐渐发展的过程中，开发者们使其不断壮大和加强。慢慢的，Apache 的开发者逐步形成了专门的开发小组，最后还成立了用来推广 Apache 和其他开源软件的基金会。目前，Apache 开源基金会已经管理了数百个开源项目，而且越来越多的大型企业或有能力的开发者，将它们开发的软件开源和共享给 Apache 基金会，让更多的人，特别是全世界的开发者们能够从中受益。图 7-3 展示了 Apache 的标识。



图 7-3 Apache 基金会

Apache 因其稳定性，得到了相当多的美誉和支持。自 1996 年 4 月以来，Apache 一直是互联网上最流行的 Web 服务器程序。1999 年，有 57% 的 Web 服务器使用它为用户提供服务；2005 年，它的市场占有率甚至接近了 70%。虽然微软、Google 等巨头推出了自己的 Web 服务程序，对 Apache 的使用率带来了一定的影响，但 Apache 在 Web 服务器领域的地位仍然是不可撼动的。

Apache 之所以能长期保持领先地位，离不开它开源的精神和实践。虽然 Apache 并不是完美的，不断有新的漏洞被发现，但得益于全球开发者的共同协作，这些漏洞总能很快地被修复。因此，Apache 的安全性比其他闭源和缺少维护的 Web 服务器程序要优秀很多。

## 7.2.2 安装 Apache

Apache 已经得到广泛使用，在主流 Linux 系统配套的应用仓库中，都能找到 Apache 的完整安装包。通过安装包进行安装，要比通过编译安装更简单，能减少安装过程中由于依赖库或者环境配置错误造成的问题，也能避免安装完成和程序运行后，出现较难排查的异常情况。

这里我们选择 Ubuntu 16.04 的镜像, 作为搭建 Apache 服务的基础镜像。我们先使用交互模式启动基于 Ubuntu 镜像的容器, 开始搭建 Apache 服务器。

```
$ sudo docker run -it --name apache ubuntu:16.04 /bin/bash
root@aeadb5ab2e3b:/#
```

接着通过 APT 软件仓库来安装 Apache 服务器程序。Apache 经过多年的发展, 目前已经推出了两个主要版本。与其他软件不同的是, Apache 的两个版本由于基础架构发生了很大的变化, 所以都是独立维护的, 因此不能将其看作简单的版本迭代。而我们在这一章所介绍的 Apache 服务器的安装, 选择了较新、较高效、较稳定的 Apache 2 进行讲解。通常我们也以 Apache 2 作为较新版本 Apache 程序的名称。APT 软件仓库只准备了较常见的新版本 Apache, 并将其命名为 apache2。在运行安装命令前, 可以使用 apt-get update 命令对当前环境中已有的程序进行更新, 以确保所有程序都处于最佳的状态。

```
root@aeadb5ab2e3b:/# apt-get update && apt-get -y install apache2
Get:1 http://archive.ubuntu.com/ubuntu xenial InRelease [247 kB]
Get:2 http://archive.ubuntu.com/ubuntu xenial-updates InRelease [95.7 kB]
Get:3 http://archive.ubuntu.com/ubuntu xenial-security InRelease [94.5 kB]
Get:4 http://archive.ubuntu.com/ubuntu xenial/main Sources [1103 kB]
Get:5 http://archive.ubuntu.com/ubuntu xenial/restricted Sources [5179 B]
Get:6 http://archive.ubuntu.com/ubuntu xenial/universe Sources [9802 kB]
Get:7 http://archive.ubuntu.com/ubuntu xenial/main amd64 Packages [1558 kB]
Get:8 http://archive.ubuntu.com/ubuntu xenial/restricted amd64 Packages [14.1 kB]
Get:9 http://archive.ubuntu.com/ubuntu xenial/universe amd64 Packages [9827 kB]
Get:10 http://archive.ubuntu.com/ubuntu xenial-updates/main Sources [230 kB]
Get:11 http://archive.ubuntu.com/ubuntu xenial-updates/universe Sources [114 kB]
Get:12 http://archive.ubuntu.com/ubuntu xenial-updates/main amd64 Packages [481 kB]
Get:13 http://archive.ubuntu.com/ubuntu xenial-updates/universe amd64 Packages [400 kB]
Get:14 http://archive.ubuntu.com/ubuntu xenial-security/main Sources [45.9 kB]
Get:15 http://archive.ubuntu.com/ubuntu xenial-security/universe Sources [9623 B]
Get:16 http://archive.ubuntu.com/ubuntu xenial-security/main amd64 Packages [168 kB]
Get:17 http://archive.ubuntu.com/ubuntu xenial-security/universe amd64 Packages [48.2 kB]
Fetched 24.2 MB in 1min 5s (369 kB/s)
Reading package lists... Done
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  apache2-bin apache2-data apache2-utils file ifupdown iproute2
```

```
isc-dhcp-client isc-dhcp-common libapr1 libaprutil1 libaprutil1-dbd-sqlite3
libaprutil1-ldap libasn1-8-heimdal libatml libdns-export162 libexpat1
libffi6 libgdbm3 libgmp10 libgnutls30 libgssapi3-heimdal libhcrypto4-heimdal
libheimbase1-heimdal libheimntlm0-heimdal libhogweed4 libhx509-5-heimdal
libicu55 libidn11 libisc-export160 libkrb5-26-heimdal libldap-2.4-2
liblua5.1-0 libmagic1 libmn10 libnettle6 libp11-kit0 libperl5.22
libroken18-heimdal libsasl2-2 libsasl2-modules libsasl2-modules-db
libsqlite3-0 libssl1.0.0 libtasn1-6 libwind0-heimdal libxml2 libxtables11
mime-support netbase openssl perl perl-modules-5.22 rename sgml-base
ssl-cert xml-core
```

Suggested packages:

```
www-browser apache2-doc apache2-suexec-pristine | apache2-suexec-custom ufw
ppp rdnsd iproute2-doc resolvconf avahi-autoipd isc-dhcp-client-ddns
apparmor gnutls-bin libsasl2-modules-otp libsasl2-modules-ldap
libsasl2-modules-sql libsasl2-modules-gssapi-mit
| libsasl2-modules-gssapi-heimdal ca-certificates perl-doc
libterm-readline-gnu-perl | libterm-readline-perl-perl make sgml-base-doc
openssl-blacklist debhelper
```

The following NEW packages will be installed:

```
apache2 apache2-bin apache2-data apache2-utils file ifupdown iproute2
isc-dhcp-client isc-dhcp-common libapr1 libaprutil1 libaprutil1-dbd-sqlite3
libaprutil1-ldap libasn1-8-heimdal libatml libdns-export162 libexpat1
libffi6 libgdbm3 libgmp10 libgnutls30 libgssapi3-heimdal libhcrypto4-heimdal
libheimbase1-heimdal libheimntlm0-heimdal libhogweed4 libhx509-5-heimdal
libicu55 libidn11 libisc-export160 libkrb5-26-heimdal libldap-2.4-2
liblua5.1-0 libmagic1 libmn10 libnettle6 libp11-kit0 libperl5.22
libroken18-heimdal libsasl2-2 libsasl2-modules libsasl2-modules-db
libsqlite3-0 libssl1.0.0 libtasn1-6 libwind0-heimdal libxml2 libxtables11
mime-support netbase openssl perl perl-modules-5.22 rename sgml-base
ssl-cert xml-core
```

0 upgraded, 57 newly installed, 0 to remove and 0 not upgraded.

Need to get 22.7 MB of archives.

After this operation, 102 MB of additional disk space will be used.

```
Get:1 http://archive.ubuntu.com/ubuntu xenial/main amd64 libatml amd64 1:2.5.1-1.5
[24.2 kB]
```

```
Get:2 http://archive.ubuntu.com/ubuntu xenial/main amd64 libmn10 amd64 1.0.3-5 [12.0
kB]
```

```
Get:3 http://archive.ubuntu.com/ubuntu xenial/main amd64 libgdbm3 amd64 1.8.3-13.1
[16.9 kB]
```

...

在 Ubuntu 或 Debian 等使用 APT 作为默认软件仓库的系统里，使用 APT 进行安装



更方便, 因为 APT 已经把 Apache 需要的所有软件都自动进行了安装。在安装过程中, 可以看到 APT 所提示的 Apache 依赖软件和库, 这些程序都会被 APT 安装或更新到系统里。当然, 使用其他比较完整的 Linux 系统时, 都有默认的软件仓库, 比如 Cent OS 和 Red Hat 的 Yum 等。在这些系统中, 我们也可以通过相应的软件仓库来完成对 Apache 的安装。

通过编译安装 Apache 也是可行的, 但是需要自己解决软件和类库依赖的问题, 并且编译操作会产生大量的中间文件, 如果不清理这些文件, 会使生成的镜像层非常庞大, 影响到使用这个镜像的容器运行时对资源的占用。所以对于已经在软件仓库中存在, 或者可以找到现有编译结果的软件, 建议使用已有的程序来完成安装。

若在 Web 网站中使用 HTTPS 协议, 就需要开启 Apache 的 TLS 安全加密模块。通过 Apache 自带的控制程序, 可以很轻松地完成这个操作。

```
root@aeadb5ab2e3b:/# a2ensite default-ssl
Enabling site default-ssl.
To activate the new configuration, you need to run:
    service apache2 reload
root@aeadb5ab2e3b:/# a2enmod ssl
Considering dependency setenvif for ssl:
Module setenvif already enabled
Considering dependency mime for ssl:
Module mime already enabled
Considering dependency socache_shmcb for ssl:
Enabling module socache_shmcb.
Enabling module ssl.
See /usr/share/doc/apache2/README.Debian.gz on how to configure SSL and create
self-signed certificates.
To activate the new configuration, you need to run:
    service apache2 restart
```

至此, Apache 就安装完成了。

### 7.2.3 构建 Apache 镜像

要使用 Dockerfile 创建 Apache 的镜像, 首先要编写一个指明 Apache 搭建过程的 Dockerfile, 用于镜像的构建。

根据我们之前在容器中进行的 Apache 服务器搭建实践, 可以很轻松地把前面用到的命令书写成对应的 Dockerfile 指令, 并组成 Dockerfile。

```

# Apache Server
# VERSION 0.0.1

# 基础镜像
FROM ubuntu:16.04
# 维护者信息
MAINTAINER You Ming <youming@funcuter.org>

# 安装 Apache
RUN apt-get update && apt-get -y install apache2 && apt-get clean

# 开启 HTTPS 支持
RUN /usr/sbin/a2ensite default-ssl
RUN /usr/sbin/a2enmod ssl

# 对外暴露 HTTP 使用的 80 端口和 HTTPS 使用的 443 端口
EXPOSE 80
EXPOSE 443

# 启动命令, 通过-D 参数切换到前台运行
CMD ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]

```

在 Dockerfile 中安装 Apache 时还执行了 `apt-get clean` 命令。这条命令用于清理安装 APT 时产生的中间文件, 使镜像层变得更小巧。如果想使用 `apt-get clean` 命令, 必须把它与安装命令放置在同一个 `RUN` 指令下。否则两个命令处理的文件会在不同的镜像层中, 会增加镜像的体积。

保存编写好的 Dockerfile, 将 Dockerfile 所在的目录传入 `docker build`, 再开始构建 Apache 镜像。

```

$ sudo docker build -t ymdot/apache ./apache
Sending build context to Docker daemon 2.048 kB
Step 1 : FROM ubuntu:16.04
----> bd3d4369aebc
Step 2 : MAINTAINER You Ming <youming@funcuter.org>
----> Running in 34028b04529b
----> e5205ead4576
Removing intermediate container 34028b04529b
Step 3 : RUN apt-get update && apt-get -y install apache2 && apt-get clean
----> Running in 33a031f2f761
...
Step 4 : RUN /usr/sbin/a2ensite default-ssl

```

```

---> Running in b3505a1714a3
Enabling site default-ssl.
To activate the new configuration, you need to run:
    service apache2 reload
---> d4e9a7195501
Removing intermediate container b3505a1714a3
Step 5 : RUN /usr/sbin/a2enmod ssl
---> Running in 09b528dea246
Considering dependency setenvif for ssl:
Module setenvif already enabled
Considering dependency mime for ssl:
Module mime already enabled
Considering dependency socache_shmcb for ssl:
Enabling module socache_shmcb.
Enabling module ssl.
See /usr/share/doc/apache2/README.Debian.gz on how to configure SSL and create self-
signed certificates.
To activate the new configuration, you need to run:
    service apache2 restart
---> 7500cd810fa1
Removing intermediate container 09b528dea246
Step 6 : EXPOSE 80
---> Running in 68e2c822ae69
---> 35fd04838a2b
Removing intermediate container 68e2c822ae69
Step 7 : EXPOSE 443
---> Running in 431e02dfbf6d
---> 24a8262a51b4
Removing intermediate container 431e02dfbf6d
Step 8 : CMD /usr/sbin/apache2ctl -D FOREGROUND
---> Running in 5f01ef0b630d
---> 326af1da920d
Removing intermediate container 5f01ef0b630d
Successfully built 326af1da920d

```

构建完成后可以在本地的镜像仓库中看到刚刚编译好的 Apache 镜像。

```

$ sudo docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
ymdot/apache        latest             326af1da920d       2 minutes ago      264.7 MB
...

```



## 7.2.4 测试 Apache 容器

我们基于刚刚构建的镜像新建容器，测试 Apache 服务程序是否搭建成功。

```
$ sudo docker run -d -P --name apache ymdot/apache
a463ecf01dba73488a30878f74259907584f8eebdb028c310a2a6126e75e1c4d
$ sudo docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	PORTS		NAMES
a463ecf01dba	ymdot/apache	"/usr/sbin/apache2ctl"	1 seconds ago
Up 1 seconds	0.0.0.0:32769->80/tcp, 0.0.0.0:32768->443/tcp		apache

启动程序后发现，Docker 为 Apache 的 HTTP 服务分配了 32769 端口作为映射端口。打开浏览器，使用这个端口访问本地主机域名 localhost。

如图 7-4 所示，浏览器中显示出了 Apache 服务程序提供的默认页面。页面中也显示 Apache 程序所运行的操作系统为 Ubuntu 系统，这表示程序运行于容器之中，其所能识别的操作系统是容器中模拟的操作系统，而非宿主机真实的操作系统。

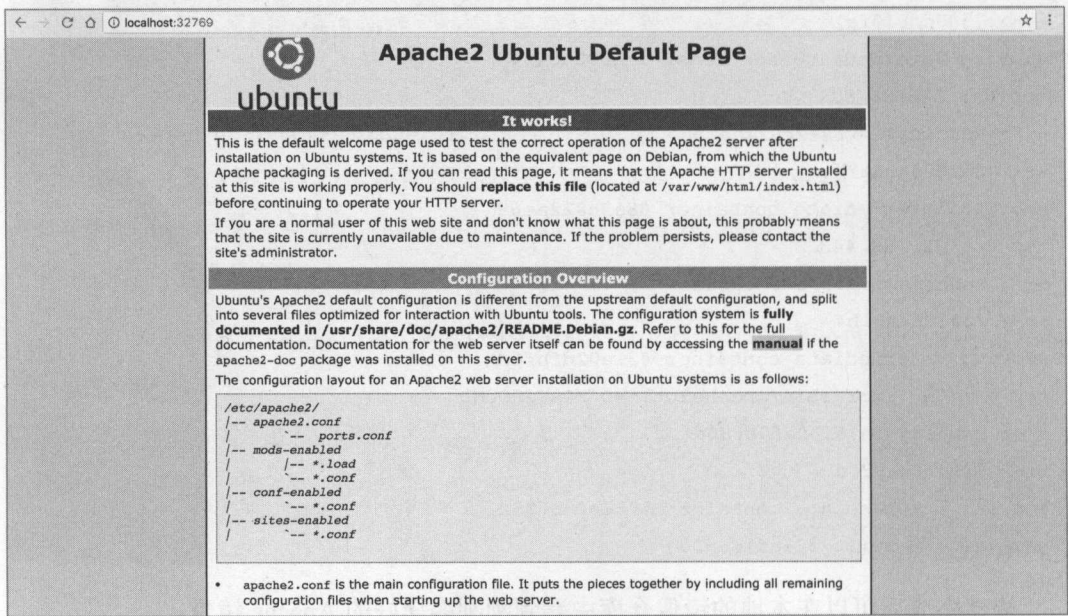


图 7-4 Apache 服务程序提供的默认页面

通过检验，含有 Apache 服务程序的 Docker 镜像文件已经搭建完成，也拥有了能够构建这个镜像的 Dockerfile 文件。通过将 Dockerfile 文件迁移到其他机器中，能很快地完成镜像的迁移。

## 7.3 Nginx

Nginx 是 Web 服务程序中以并发处理能力著称的优秀软件，也是目前发展速度最快的 Web 服务程序。

### 7.3.1 关于 Nginx

与 Apache 一样，Nginx 也是一个 Web 服务器程序。Nginx 最初由俄国的搜索引擎网站 Rambler 使用，之后遵循 BSD-like 协议进行了开源。由于卓越的并发处理性能，其在 Web 服务器程序中占有一定的地位及市场占有率。图 7-5 展示了 Nginx 的标识。



图 7-5 Nginx

与 Apache 不同，Nginx 是一款面向性能设计的 Web 服务器程序。绝大多数的处理过程，Nginx 是通过反向代理的方式交给指定的程序进行的，也就是说，Nginx 本身是一个轻量级的 Web 服务器。Nginx 充分发挥了多核 CPU 的处理能力，并利用异步处理逻辑减少了调度过程中的开销，而且其他部分的优化也让 Nginx 在并发处理上有着巨大的优势。特别是在 Linux 系统中，Nginx 可以使用 epoll 这类 IO 复用事件模型，所以在 Linux 系统里 Nginx 有更高的处理能力。

因为主攻的方向是高并发，所以 Nginx 将动态处理的过程交给了其他程序来完成，其只专心做请求的接受和响应。这使 Nginx 不但占用的内存等计算机资源少，而且稳定性大幅提高。对于承接处理任务的动态处理程序，Nginx 也提供了众多支持方案，包括 HTTP、HTTPS、SMTP、POP3、IMAP 在内的许多协议，Nginx 都可以反向代理通过它们的请求。

完全模块化也是 Nginx 的一大特点。在 Nginx 中没有主体的概念，即使 HTTP 也是以模块的形式存在的。

Nginx 本身的特性及其开源策略，造就了许多基于 Nginx 的 Web 服务器。其中有以高并发处理优化著称的 Tengine，也有内置 Lua 脚本处理层的 OpenResty。可以说，在 Apache 之外，Nginx 为 Web 服务器领域又提供了一片天空。

## 7.3.2 安装 Nginx

因为 APT 软件仓库中已经存在编译完成且可以直接运行的 Nginx 程序，所以可以直接使用 APT 仓库中的 Nginx 进行 Web 服务器程序的搭建。这里以 APT 软件仓库作为默认软件仓库的 Debian Linux 系统镜像，用来当作即将搭建的 Nginx 镜像的基础镜像。我们选择 Debian 最新的稳定版本 Jessie，启动基于这个基础镜像的容器，并进入到容器中。

```
$ sudo docker run -it --name nginx debian:jessie /bin/bash
root@laae46290670:/#
```

在通过 APT 软件仓库安装之前，要对 APT 软件仓库进行更新。

```
root@laae46290670:/# apt-get update
Get:1 http://security.debian.org jessie/updates InRelease [63.1 kB]
Ign http://httpredir.debian.org jessie InRelease
Get:2 http://httpredir.debian.org jessie-updates InRelease [142 kB]
Get:3 http://httpredir.debian.org jessie Release.gpg [2373 B]
Get:4 http://httpredir.debian.org jessie Release [148 kB]
Get:5 http://security.debian.org jessie/updates/main amd64 Packages [387 kB]
Get:6 http://httpredir.debian.org jessie-updates/main amd64 Packages [17.6 kB]
Get:7 http://httpredir.debian.org jessie/main amd64 Packages [9032 kB]
...
```

因为 Nginx 以模块化设计，所以可以很容易地将不同功能的模块进行组合，以达到我们的需求。虽然 APT 中的 Nginx 程序只默认打包了少数几个核心模块，但是 APT 会将很多未安装模块的程序或它们的依赖程序推荐给用户，并一起安装。为了降低镜像层的容量，在安装 Nginx 时通过 APT 中的 `--no-install-recommends` 和 `--no-install-suggests` 两个参数来禁止推荐程序的安装。

```
root@laae46290670:/# apt-get install --no-install-recommends --no-install-suggests
-y nginx
Reading package lists... Done
Building dependency tree... Done
The following extra packages will be installed:
  fontconfig-config fonts-dejavu-core init-system-helpers libexpat1 libfontconfig1
libfreetype6 libgd3
  libgeoip1 libjbig0 libjpeg62-turbo libpng12-0 libssl1.0.0 libtiff5 libvpx1 libx11-6
libx11-data
  libxau6 libxcb1 libxdmcp6 libxml2 libxpm4 libxslt1.1 nginx-common nginx-full ucf
Suggested packages:
  libgd-tools geoip-bin fcgiwrap nginx-doc ssl-cert
Recommended packages:
```



```

geoip-database xml-core
The following NEW packages will be installed:
  fontconfig-config fonts-dejavu-core init-system-helpers libexpat1 libfontconfig1
libfreetype6 libgd3
  libgeoip1 libjbig0 libjpeg62-turbo libpng12-0 libssl1.0.0 libtiff5 libvpx1 libx11-6
libx11-data
  libxau6 libxcb1 libxdmcp6 libxml2 libxpm4 libxslt1.1 nginx nginx-common nginx-full ucf
0 upgraded, 26 newly installed, 0 to remove and 5 not upgraded.
Need to get 7311 kB of archives.
After this operation, 19.8 MB of additional disk space will be used.
Get:1 http://security.debian.org/ jessie/updates/main libxml2 amd64 2.9.1+dfsg1-
5+deb8u2 [802 kB]
Get:2 http://security.debian.org/ jessie/updates/main libexpat1 amd64 2.1.0-6+deb8u3
[80.0 kB]
Get:3 http://security.debian.org/ jessie/updates/main fontconfig-config all 2.11.0-
6.3+deb8u1 [274 kB]
Get:4 http://security.debian.org/ jessie/updates/main libfontconfig1 amd64 2.11.0-
6.3+deb8u1 [329 kB]
Get:5 http://security.debian.org/ jessie/updates/main libgd3 amd64 2.1.0-5+deb8u6
[148 kB]
Get:6 http://security.debian.org/ jessie/updates/main libxslt1.1 amd64 1.1.28-2+
deb8u1 [232 kB]
Get:7 http://security.debian.org/ jessie/updates/main nginx-common all 1.6.2-5+
deb8u2 [86.7 kB]
Get:8 http://security.debian.org/ jessie/updates/main nginx-full amd64 1.6.2-5+
deb8u2+b1 [430 kB]
Get:9 http://security.debian.org/ jessie/updates/main nginx all 1.6.2-5+deb8u2 [72.2
kB]
Get:10 http://httpredir.debian.org/debian/ jessie/main libssl1.0.0 amd64 1.0.1t-1+
deb8u2 [1045 kB]
Get:11 http://httpredir.debian.org/debian/ jessie/main libssl1.0.0 amd64 1.0.1t-1+
deb8u2 [1045 kB]
...

```

因为可能会在 Nginx 中使用到 HTTPS, 所以还需要安装 CA 证书来提供相关的支持。

```

root@laae46290670:/# apt-get install -y ca-certificates
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following extra packages will be installed:
  openssl
The following NEW packages will be installed:

```

```

ca-certificates openssl
0 upgraded, 2 newly installed, 0 to remove and 5 not upgraded.
Need to get 868 kB of archives.
After this operation, 1500 kB of additional disk space will be used.
Get:1 http://httpredir.debian.org/debian/ jessie/main openssl amd64 1.0.1t-1+deb8u2
[664 kB]
Get:2 http://httpredir.debian.org/debian/ jessie/main ca-certificates all 20141019+
deb8u1 [204 kB]
Fetched 868 kB in 17s (49.8 kB/s)
debconf: delaying package configuration, since apt-utils is not installed
Selecting previously unselected package openssl.
(Reading database ... 8155 files and directories currently installed.)
Preparing to unpack .../openssl_1.0.1t-1+deb8u2_amd64.deb ...
Unpacking openssl (1.0.1t-1+deb8u2) ...
Selecting previously unselected package ca-certificates.
Preparing to unpack .../ca-certificates_20141019+deb8u1_all.deb ...
Unpacking ca-certificates (20141019+deb8u1) ...
Setting up openssl (1.0.1t-1+deb8u2) ...
Setting up ca-certificates (20141019+deb8u1) ...
debconf: unable to initialize frontend: Dialog
debconf: (No usable dialog-like program is installed, so the dialog based frontend
cannot be used. at /usr/share/perl5/Debconf/FrontEnd/Dialog.pm line 76.)
debconf: falling back to frontend: Readline
debconf: unable to initialize frontend: Readline
debconf: (Can't locate Term/ReadLine.pm in @INC (you may need to install the
Term::ReadLine module) (@INC contains: /etc/perl /usr/local/lib/x86_64-linux-gnu/
perl/5.20.2 /usr/local/share/perl/5.20.2 /usr/lib/x86_64-linux-gnu/perl5/5.20 /usr/
share/perl5 /usr/lib/x86_64-linux-gnu/perl/5.20 /usr/share/perl/5.20 /usr/local/
lib/site_perl .) at /usr/share/perl5/Debconf/FrontEnd/Readline.pm line 7.)
debconf: falling back to frontend: Teletype
Processing triggers for ca-certificates (20141019+deb8u1) ...
Updating certificates in /etc/ssl/certs... 174 added, 0 removed; done.
Running hooks in /etc/ca-certificates/update.d....done.

```

至此，我们就成功地在 Docker 中安装了 Nginx 服务程序。

### 7.3.3 构建 Nginx 镜像

根据实践中总结的用于搭建 Nginx 服务器的命令，总结出可以构建 Nginx 镜像的 Dockerfile。



```

# Nginx Server
# VERSION 0.0.1

# 基础镜像
FROM debian:jessie

# 维护者信息
MAINTAINER You Ming <youming@funcuter.org>

# 安装 Nginx
RUN apt-get update && apt-get install --no-install-recommends --no-install-suggests
-y ca-certificates nginx

# 对外暴露 HTTP 使用的 80 端口和 HTTPS 使用的 443 端口
EXPOSE 80 443

# 启动命令, 通过 -g 参数修改配置, 让 Nginx 使用前台运行模式
CMD ["nginx", "-g", "daemon off;"]

```

将 Dockerfile 的内容保存到文件中, 并通过 `docker build` 命令构建 Dockerfile 中定义的 Nginx 镜像。

```

$ sudo docker build -t ymdot/nginx ./nginx
Sending build context to Docker daemon 2.048 kB
Step 1 : FROM debian:jessie
----> 1b088884749b
Step 2 : MAINTAINER You Ming <youming@funcuter.org>
----> Using cache
----> ef8c182cccfa
Step 3 : RUN apt-get update && apt-get install --no-install-recommends --no-install-
suggests -y ca-certificates nginx
----> Running in 6d498f7f98dc
...
----> 19de6aee4e9c
Removing intermediate container 6d498f7f98dc
Step 4 : EXPOSE 80 443
----> Running in 5d81634dc23a
----> 7166acded696
Removing intermediate container 5d81634dc23a
Step 5 : CMD nginx -g daemon off;
----> Running in 8d9cf39cd57c
----> ed112714fe95
Removing intermediate container 8d9cf39cd57c
Successfully built ed112714fe95

```



终端显示出构建完成的信息，我们就能从本地的镜像仓库里找到刚刚构建的 Nginx 镜像了。

```
$ sudo docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
ymdot/nginx	latest	ed112714fe95	34 minutes ago	157.3 MB

### 7.3.4 测试 Nginx 镜像

镜像构建完成之后，通过 Nginx 镜像新建和运行一个 Nginx 容器，检查搭建的 Nginx 服务器是否能够正常运行。

```
$ sudo docker rrun -d --name nginx -P ymdot/nginx
cca691a21ceaa44314e7bc730061f5195afdc43af252e06c8adb07469a5cd371
$ sudo docker port nginx
443/tcp -> 0.0.0.0:32768
80/tcp -> 0.0.0.0:32769
```

从浏览器中打开本地地址，如图 7-6 所示，可以看到 Nginx 默认访问页面已经出现，这说明制作的 Nginx 容器已经能够正常运行了。

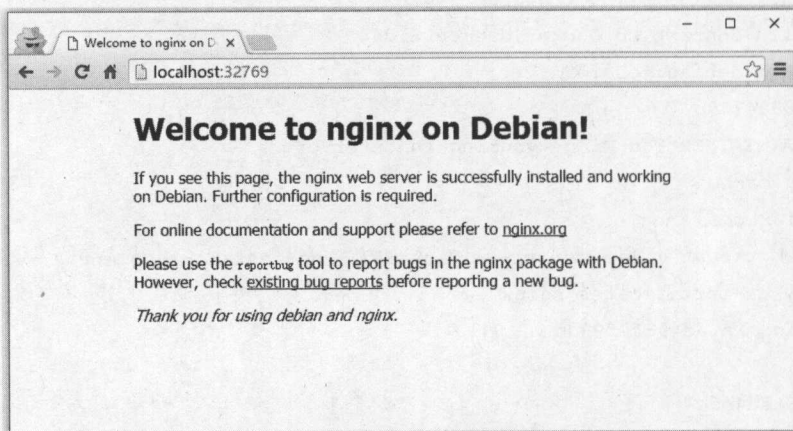


图 7-6 Nginx 默认访问页面

## 7.4 Tomcat

Tomcat 也是常见的 Web 服务器之一，与 Apache 和 Nginx 不同，Tomcat 主要针对的是基于 Java 开发的 Web 应用程序。

## 7.4.1 Tomcat 简介

对于 Web 服务来说,动态处理是向用户提供更好体验的绝佳方式,用户提供给服务器的信息,通过动态处理程序能够被很好地处理和分析。在目前主流的 Web 服务器端动态处理程序中,使用最多的编程语言当属 Java 和 PHP。对于 PHP 来说,由于它和很多 Apache、Nginx 等常用的 Web 服务器都实现了通用网关接口(Common Gateway Interface)所规定的通信方式。所以这些 Web 服务器程序能够直接将客户端传来的请求传递给 PHP 程序进行处理。而对于 Java 来说,我们习惯将处理动态请求的程序封装在 Servlet (Server Applet) 中,这就需要专门实现过 Servlet 的程序来运转它们。Tomcat 就是这样一个支持 Java 程序处理的 Servlet 程序。图 7-7 展示了 Tomcat 的标识。

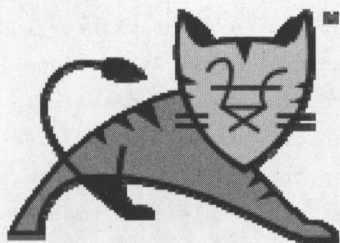


图 7-7 Tomcat 的标识

Tomcat 也由 Apache 基金会进行管理和维护,能够与 Apache 服务器进行协作,处理 Apache 服务器收到的动态请求,也能够独立运行成 Web 服务器,自己处理 HTTP 请求。

在 Java Web 界, Tomcat 是非常重要的 Web 服务器,也是几乎所有学习和使用 Java 开发 Web 服务程序的开发者需要掌握的基础知识。

## 7.4.2 安装 Tomcat

因为 Tomcat 是服务于 Java Web 的程序,在运行的过程中离不开 Java,所以在搭建和安装 Tomcat 时,也需要提供 Java 的运行环境。在以往的服务器搭建过程中,通常是分别安装 Java 和 Tomcat 程序,但有了 Docker 之后,就该好好利用它独特的优势。在 Docker 的镜像仓库中有世界各地的开发者共享的已经安装好的 Java 镜像,只要基于这些镜像搭建 Tomcat,就能省去安装和配置 Java 的过程。

Docker 官方提供的 Java 镜像有很多版本和类型的 Java 程序可供选择,我们只需要 Java 的运行环境,所以使用 8-jre 标签下的镜像作为搭建 Tomcat 的基础镜像。java:8-jre 镜像包含了基于 Java 8 的运行环境,方便我们使用。

首先，基于 Java 镜像建立容器，并进入到容器之中。

```
$ sudo docker run -it --name tomcat java:8-jre /bin/bash
root@8969222e3e5a:/#
```

由于使用的 Java 中内置了 APT 软件仓库，所以可以通过 APT 来安装 Tomcat，过程非常简单。目前，Tomcat 8 是最新的稳定版本，而在 APT 软件仓库中，对应的软件名为 tomcat 8，所以我们通过这个名称来安装 Tomcat。

```
root@8969222e3e5a:/# apt-get update
Get:1 http://security.debian.org jessie/updates InRelease [63.1 kB]
Get:2 http://security.debian.org jessie/updates/main amd64 Packages [387 kB]
Ign http://httpredir.debian.org jessie InRelease
Get:3 http://httpredir.debian.org jessie-updates InRelease [142 kB]
Get:4 http://httpredir.debian.org jessie-backports InRelease [166 kB]
Get:5 http://httpredir.debian.org jessie Release.gpg [2373 B]
Get:6 http://httpredir.debian.org jessie-updates/main amd64 Packages [17.6 kB]
Get:7 http://httpredir.debian.org jessie Release [148 kB]
Get:8 http://httpredir.debian.org jessie-backports/main amd64 Packages [883 kB]
Get:9 http://httpredir.debian.org jessie/main amd64 Packages [9032 kB]
Fetched 10.8 MB in 1min 26s (126 kB/s)
Reading package lists... Done
root@8969222e3e5a:/# apt-get install -y tomcat8
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following extra packages will be installed:
  authbind libcommons-dbcj-java libcommons-pool-java libecj-java libtomcat8-java
  tomcat8-common
Suggested packages:
  libcommons-dbcj-java-doc libgeronimo-jta-1.1-spec-java ecj ant libecj-java-gcj
  libtcnative-1
  tomcat8-admin tomcat8-docs tomcat8-examples tomcat8-user
The following NEW packages will be installed:
  authbind libcommons-dbcj-java libcommons-pool-java libecj-java libtomcat8-java
  tomcat8
  tomcat8-common
0 upgraded, 7 newly installed, 0 to remove and 1 not upgraded.
Need to get 6614 kB of archives.
After this operation, 8398 kB of additional disk space will be used.
Get:1 http://security.debian.org/ jessie/updates/main libtomcat8-java all 8.0.14-1+
deb8u2 [4585 kB]
Get:2 http://httpredir.debian.org/debian/ jessie/main authbind amd64 2.1.1 [20.2 kB]
```



```
Get:3 http://httpredir.debian.org/debian/ jessie/main libcommons-pool-java all 1.6-2
[105 kB]
Get:4 http://httpredir.debian.org/debian/ jessie/main libcommons-dbcj-java all 1.4-5
[155 kB]
Get:5 http://httpredir.debian.org/debian/ jessie/main libecj-java all 3.10.1-1 [1647 kB]
Get:6 http://security.debian.org/ jessie/updates/main tomcat8-common all 8.0.14-1+
deb8u2 [55.9 kB]
Get:7 http://security.debian.org/ jessie/updates/main tomcat8 all 8.0.14-1+deb8u2
[45.1 kB]
Fetched 6614 kB in 1min 14s (88.9 kB/s)
debconf: delaying package configuration, since apt-utils is not installed
Selecting previously unselected package authbind.
(Reading database ... 9174 files and directories currently installed.)
Preparing to unpack .../authbind_2.1.1_amd64.deb ...
Unpacking authbind (2.1.1) ...
Selecting previously unselected package libcommons-pool-java.
Preparing to unpack .../libcommons-pool-java_1.6-2_all.deb ...
Unpacking libcommons-pool-java (1.6-2) ...
Selecting previously unselected package libcommons-dbcj-java.
Preparing to unpack .../libcommons-dbcj-java_1.4-5_all.deb ...
Unpacking libcommons-dbcj-java (1.4-5) ...
Selecting previously unselected package libecj-java.
Preparing to unpack .../libecj-java_3.10.1-1_all.deb ...
Unpacking libecj-java (3.10.1-1) ...
Selecting previously unselected package libtomcat8-java.
Preparing to unpack .../libtomcat8-java_8.0.14-1+deb8u2_all.deb ...
Unpacking libtomcat8-java (8.0.14-1+deb8u2) ...
Selecting previously unselected package tomcat8-common.
Preparing to unpack .../tomcat8-common_8.0.14-1+deb8u2_all.deb ...
Unpacking tomcat8-common (8.0.14-1+deb8u2) ...
Selecting previously unselected package tomcat8.
Preparing to unpack .../tomcat8_8.0.14-1+deb8u2_all.deb ...
Unpacking tomcat8 (8.0.14-1+deb8u2) ...
Processing triggers for systemd (215-17+deb8u4) ...
Setting up authbind (2.1.1) ...
Setting up libcommons-pool-java (1.6-2) ...
Setting up libcommons-dbcj-java (1.4-5) ...
Setting up libecj-java (3.10.1-1) ...
Setting up libtomcat8-java (8.0.14-1+deb8u2) ...
Setting up tomcat8-common (8.0.14-1+deb8u2) ...
Setting up tomcat8 (8.0.14-1+deb8u2) ...
debconf: unable to initialize frontend: Dialog
```

```
debconf: (No usable dialog-like program is installed, so the dialog based frontend
cannot be used. at /usr/share/perl5/Debconf/FrontEnd/Dialog.pm line 76.)
debconf: falling back to frontend: Readline
debconf: unable to initialize frontend: Readline
debconf: (Can't locate Term/ReadLine.pm in @INC (you may need to install the
Term::ReadLine module) (@INC contains: /etc/perl /usr/local/lib/x86_64-linux-gnu/
perl/5.20.2 /usr/local/share/perl/5.20.2 /usr/lib/x86_64-linux-gnu/perl5/5.20 /usr/
share/perl5 /usr/lib/x86_64-linux-gnu/perl/5.20 /usr/share/perl/5.20 /usr/local/
lib/site_perl .) at /usr/share/perl5/Debconf/FrontEnd/Readline.pm line 7.)
debconf: falling back to frontend: Teletype

Creating config file /etc/default/tomcat8 with new version
Adding system user `tomcat8' (UID 105) ...
Adding new user `tomcat8' (UID 105) with group `tomcat8' ...
Not creating home directory `/usr/share/tomcat8'.

Creating config file /etc/logrotate.d/tomcat8 with new version
invoke-rc.d: policy-rc.d denied execution of start.
Processing triggers for systemd (215-17+deb8u4) ...
```

### 7.4.3 构建 Tomcat 镜像

根据搭建 Tomcat 程序的经验，写出构建 Tomcat 镜像的 Dockerfile。

```
# Tomcat Server
# VERSION 0.0.1

# 基础镜像
FROM java:8-jre

# 维护者信息
MAINTAINER You Ming <youming@funcuter.org>

# 安装 Tomcat
RUN apt-get update && apt-get install -y tomcat8

# 对外暴露 Tomcat 的默认端口
EXPOSE 8080

# 启动命令，通过-g 参数修改配置，让 Tomcat 使用前台运行模式
CMD ["/usr/share/tomcat8/bin/catalina.sh", "run"]
```

将 Dockerfile 的内容保存到文件中，并通过 `docker build` 命令尝试构建 Dockerfile 中所定义的 Tomcat 镜像。

```
$ sudo docker build -t ymdot/tomcat ./tomcat
Sending build context to Docker daemon 2.048 kB
Step 1 : FROM java:8-jre
----> 042c17968c65
Step 2 : MAINTAINER You Ming <youming@funcuter.org>
----> Running in d1587d35207d
----> 56051d113ac2
Removing intermediate container d1587d35207d
Step 3 : RUN apt-get update && apt-get install -y tomcat8
----> Running in a50108912301
...
Processing triggers for systemd (215-17+deb8u4) ...
----> 0ad606a95e39
Removing intermediate container a50108912301
Step 4 : EXPOSE 8080
----> Running in e012089f4026
----> 106161ebafd7
Removing intermediate container e012089f4026
Step 5 : CMD /usr/share/tomcat8/bin/catalina.sh run
----> Running in e142622d2a70
----> d3d468901150
Removing intermediate container e142622d2a70
Successfully built d3d468901150
```

由此发现，Tomcat 镜像的构建过程比 Apache、Nginx 要快很多，因为大多数的依赖程序都已经被包含在了基础的 Java 镜像中。所以，在使用 Docker 时，要善于使用已有的镜像，这能够很好地提高效率。

```
$ sudo docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
ymdot/tomcat	latest	d3d468901150	10 minutes ago	331.8 MB
...				

## 7.5 本章小结

Web 服务是最常用的通过互联网向用户提供服务的方式，也是绝大多数互联网应用不可或缺的“脸面”。搭建、配置及维护 Web 服务，是开发和运维中常见的任务。



在本章中，我们了解了 Web 服务的基本知识，对 Apache、Nginx、Tomcat 几个常见的 Web 服务程序在 Docker 中的安装和使用进行了实践。利用 Docker 能够更快地部署 Web 应用，其中包括对提供 Web 服务的部署。通过本章的学习，我们能够更好地掌握 Docker 在 Web 应用中使用的场景，以及利用 Docker 进行 Web 服务的迁移，特别是 Web 服务程序的迁移过程。

# 第 8 章

## 数据库程序

数据是重要的计算资源，也是信息的载体，在应用程序中，数据既是用于计算、生产的原料，也是得到的产品。通常来说，设计程序就是处理一些给出的数据，再根据它们和我们的规则得出另一些数据。

数据的存储则是基础中的基础，在常用的数据存储方式中，存放于硬盘或者其他持久存储器是保存数据最佳的方式。而在持久化存放中，有直接存储、压缩存储、数据库存储等方式。

数据库作为专门的管理软件，在数据的增、删、改、查、共享，以及减少数据冗余度等方面，都有针对性的设计和实现。在数据量特别庞大的时候，数据库的重要作用就越能体现出来。

数据库是绝大多数程序都具备的，也是服务体系中不可或缺的一部分，主要分为关系型数据库和非关系型数据库两类。在本章中，我们就通过两种具有代表性的数据库 MySQL 和 MongoDB，来学习如何在 Docker 中搭建和使用数据库。

### 8.1 MySQL

MySQL 是众多开发者都熟悉的开源数据库，也是目前使用范围最广的开源关系型数据库。在很多应用体系中，MySQL 都是数据存储的核心部分。

### 8.1.1 MySQL 简介

在关系型数据库领域，MySQL 是当下最流行的开源数据库软件。我们所熟知的 LAMP 架构中的 M，主要指的就是 MySQL。MySQL 最初由瑞典的 MySQLAB 公司开发和维护，后来该公司被 Sun 公司收购，并列入 Sun 公司旗下。2009 年，Oracle（甲骨文）公司收购 Sun 公司，MySQL 成为 Oracle 公司旗下产品。图 8-1 展示了 MySQL 的标识。



图 8-1 MySQL 的标识

虽然和 Oracle、SQL Server 等大型数据库相比，MySQL 规模较小且功能有限，但这丝毫不影响 MySQL 受追捧的程度。特别是对于中小型企业和个人开发者来说，MySQL 所提供的功能已经绰绰有余。而且，作为开源且免费的软件，MySQL 可以大幅降低用户的使用成本。

### 8.1.2 安装 MySQL

安装 MySQL 前要建立一个容器包裹 MySQL 程序，我们选择 Debian 系统镜像作为安装 MySQL 所使用容器的基础镜像。我们基于 Debian 镜像创建一个新的容器，并进入到容器中。

```
$ sudo docker run -it --name mysql debian:jessie /bin/bash
root@99ac0c36e01c:/#
```

MySQL 有多个版本，开源且免费的是 MySQL 的社区版，也是我们要安装的版本。目前 MySQL 社区版的最新版本为 5.7.15。

因为使用 Debian 作为 MySQL 的运行系统，所以使用 Debian 系统镜像里含有的 APT 软件仓库进行 MySQL 的安装。为了安装最新的 MySQL，需要对 APT 软件仓库中的 MySQL 软件源进行配置。下面先将 MySQL 的源地址加入 APT 软件源列表配置中。

```
root@99ac0c36e01c:/# echo "deb http://repo.mysql.com/apt/debian/ jessie mysql-5.7"
> /etc/apt/sources.list.d/mysql.list
```



除了添加源地址,为了保证软件的安全性和完整性,APT 还会检查软件包的签名。所以,需要把 MySQL 的软件包签名加入到 APT 中。

```
root@99ac0c36e01c:/# apt-key adv --keyserver ha.pool.sks-keyservers.net --recv-keys
A4A9406876FCBD3C456770C88C718D3B5072E1F5
Executing: gpg --ignore-time-conflict --no-options --no-default-keyring --homedir
/tmp/tmp.ei9fmPhNem --no-auto-check-trustdb --trust-model always --primary-keyring
/etc/apt/trusted.gpg --keyring /etc/apt/trusted.gpg.d/debian-archive-jessie-automatic.
gpg --keyring /etc/apt/trusted.gpg.d/debian-archive-jessie-security-automatic.gpg
--keyring /etc/apt/trusted.gpg.d/debian-archive-jessie-stable.gpg --keyring /etc/
apt/trusted.gpg.d/debian-archive-squeeze-automatic.gpg --keyring /etc/apt/trusted.
gpg.d/debian-archive-squeeze-stable.gpg --keyring /etc/apt/trusted.gpg.d/debian-
archive-wheezy-automatic.gpg --keyring /etc/apt/trusted.gpg.d/debian-archive-wheezy-
stable.gpg --keyserver ha.pool.sks-keyservers.net --recv-keys A4A9406876FCBD3C456
770C88C718D3B5072E1F5
gpg: requesting key 5072E1F5 from hkp server ha.pool.sks-keyservers.net
gpg: key 5072E1F5: public key "MySQL Release Engineering <mysql-build@oss.oracle.com>"
imported
gpg: Total number processed: 1
gpg:             imported: 1
```

再进行 APT 仓库的更新,更新软件及下载源等信息。

```
root@99ac0c36e01c:/# apt-get update
Get:1 http://repo.mysql.com jessie InRelease [23.9 kB]
Get:2 http://repo.mysql.com jessie/mysql-5.7 amd64 Packages [2721 B]
Hit http://security.debian.org jessie/updates InRelease
Get:3 http://security.debian.org jessie/updates/main amd64 Packages [390 kB]
Ign http://httpredir.debian.org jessie InRelease
Hit http://httpredir.debian.org jessie-updates InRelease
Hit http://httpredir.debian.org jessie Release.gpg
Get:4 http://httpredir.debian.org jessie-updates/main amd64 Packages [17.6 kB]
Hit http://httpredir.debian.org jessie Release
Get:5 http://httpredir.debian.org jessie/main amd64 Packages [9064 kB]
Fetched 9498 kB in 11min 50s (13.4 kB/s)
Reading package lists... Done
```

先通过 APT 安装 perl 和 pwgen 两个软件,因为进行 MySQL 数据库初始化的时候要用到这两个软件。

```
root@99ac0c36e01c:/# apt-get install -y --no-install-recommends perl pwgen
Reading package lists... Done
Building dependency tree
```

```
Reading state information... Done
The following extra packages will be installed:
  libgdbm3 perl-modules
Suggested packages:
  perl-doc libterm-readline-gnu-perl libterm-readline-perl-perl make libb-lint-perl
  libcpanplus-dist-build-perl
  libcpanplus-perl libfile-checktree-perl liblog-message-simple-perl liblog-message-
  perl libobject-accessor-perl
Recommended packages:
  rename libarchive-extract-perl libmodule-pluggable-perl libpod-latex-perl libterm-
  ui-perl libtext-soundex-perl
  libcgi-pm-perl libmodule-build-perl libpackage-constants-perl
The following NEW packages will be installed:
  libgdbm3 perl perl-modules pwgen
0 upgraded, 4 newly installed, 0 to remove and 16 not upgraded.
Need to get 5232 kB of archives.
After this operation, 33.6 MB of additional disk space will be used.
Get:1 http://httpredir.debian.org/debian/ jessie/main libgdbm3 amd64 1.8.3-13.1 [30.0 kB]
Get:2 http://httpredir.debian.org/debian/ jessie/main perl-modules all 5.20.2-3+
deb8u6 [2547 kB]
Get:3 http://httpredir.debian.org/debian/ jessie/main perl amd64 5.20.2-3+deb8u6
[2637 kB]
Get:4 http://httpredir.debian.org/debian/ jessie/main pwgen amd64 2.07-1 [18.1 kB]
Fetched 5232 kB in 14min 43s (5919 B/s)
debconf: delaying package configuration, since apt-utils is not installed
Selecting previously unselected package libgdbm3:amd64.
(Reading database ... 7548 files and directories currently installed.)
Preparing to unpack .../libgdbm3_1.8.3-13.1_amd64.deb ...
Unpacking libgdbm3:amd64 (1.8.3-13.1) ...
Selecting previously unselected package perl-modules.
Preparing to unpack .../perl-modules_5.20.2-3+deb8u6_all.deb ...
Unpacking perl-modules (5.20.2-3+deb8u6) ...
Selecting previously unselected package perl.
Preparing to unpack .../perl_5.20.2-3+deb8u6_amd64.deb ...
Unpacking perl (5.20.2-3+deb8u6) ...
Selecting previously unselected package pwgen.
Preparing to unpack .../pwgen_2.07-1_amd64.deb ...
Unpacking pwgen (2.07-1) ...
Setting up libgdbm3:amd64 (1.8.3-13.1) ...
Setting up perl-modules (5.20.2-3+deb8u6) ...
Setting up perl (5.20.2-3+deb8u6) ...
update-alternatives: using /usr/bin/prename to provide /usr/bin/rename (rename) in
```



```

auto mode
Setting up pwgen (2.07-1) ...
Processing triggers for libc-bin (2.19-18+deb8u4) ...

```

接着就可以开始通过 APT 安装 MySQL 了。

```

root@99ac0c36e01c:/# apt-get install -y mysql-server="5.7.15-1debian8"
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following extra packages will be installed:
  init-system-helpers libaiol libmecab2 libnumal mysql-client mysql-common mysql-
community-client
  mysql-community-server psmisc
The following NEW packages will be installed:
  init-system-helpers libaiol libmecab2 libnumal mysql-client mysql-common mysql-
community-client
  mysql-community-server mysql-server psmisc
0 upgraded, 10 newly installed, 0 to remove and 16 not upgraded.
Need to get 420 kB/29.2 MB of archives.
After this operation, 222 MB of additional disk space will be used.
Get:1 http://httpredir.debian.org/debian/ jessie/main libaiol amd64 0.3.110-1 [9312 B]
Get:2 http://httpredir.debian.org/debian/ jessie/main libnumal amd64 2.0.10-1 [32.5 kB]
Get:3 http://httpredir.debian.org/debian/ jessie/main psmisc amd64 22.21-2 [119 kB]
Get:4 http://httpredir.debian.org/debian/ jessie/main libmecab2 amd64 0.996-1.1 [245 kB]
Get:5 http://httpredir.debian.org/debian/ jessie/main init-system-helpers all 1.22
[14.0 kB]
Fetched 420 kB in 4s (103 kB/s)
debconf: delaying package configuration, since apt-utils is not installed
Selecting previously unselected package libaiol:amd64.
(Reading database ... 8758 files and directories currently installed.)
Preparing to unpack .../libaiol_0.3.110-1_amd64.deb ...
Unpacking libaiol:amd64 (0.3.110-1) ...
Selecting previously unselected package libnumal:amd64.
Preparing to unpack .../libnumal_2.0.10-1_amd64.deb ...
Unpacking libnumal:amd64 (2.0.10-1) ...
Selecting previously unselected package mysql-common.
Preparing to unpack .../mysql-common_5.7.15-1debian8_amd64.deb ...
Unpacking mysql-common (5.7.15-1debian8) ...
Selecting previously unselected package mysql-community-client.
Preparing to unpack .../mysql-community-client_5.7.15-1debian8_amd64.deb ...
Unpacking mysql-community-client (5.7.15-1debian8) ...
Selecting previously unselected package mysql-client.

```



```
Preparing to unpack .../mysql-client_5.7.15-1debian8_amd64.deb ...
Unpacking mysql-client (5.7.15-1debian8) ...
Selecting previously unselected package psmisc.
Preparing to unpack .../psmisc_22.21-2_amd64.deb ...
Unpacking psmisc (22.21-2) ...
Selecting previously unselected package libmecab2.
Preparing to unpack .../libmecab2_0.996-1.1_amd64.deb ...
Unpacking libmecab2 (0.996-1.1) ...
Selecting previously unselected package init-system-helpers.
Preparing to unpack .../init-system-helpers_1.22_all.deb ...
Unpacking init-system-helpers (1.22) ...
Selecting previously unselected package mysql-community-server.
Preparing to unpack .../mysql-community-server_5.7.15-1debian8_amd64.deb ...
Unpacking mysql-community-server (5.7.15-1debian8) ...
Selecting previously unselected package mysql-server.
Preparing to unpack .../mysql-server_5.7.15-1debian8_amd64.deb ...
Unpacking mysql-server (5.7.15-1debian8) ...
Processing triggers for systemd (215-17+deb8u4) ...
Setting up libaiol:amd64 (0.3.110-1) ...
Setting up libnumal:amd64 (2.0.10-1) ...
Setting up mysql-common (5.7.15-1debian8) ...
update-alternatives: using /etc/mysql/my.cnf.fallback to provide /etc/mysql/my.cnf
(my.cnf) in auto mode
Setting up mysql-community-client (5.7.15-1debian8) ...
Setting up mysql-client (5.7.15-1debian8) ...
Setting up psmisc (22.21-2) ...
Setting up libmecab2 (0.996-1.1) ...
Setting up init-system-helpers (1.22) ...
Setting up mysql-community-server (5.7.15-1debian8) ...
debconf: unable to initialize frontend: Dialog
debconf: (No usable dialog-like program is installed, so the dialog based frontend
cannot be used. at /usr/share/perl5/Debconf/FrontEnd/Dialog.pm line 76.)
debconf: falling back to frontend: Readline
Configuring mysql-community-server
-----

Data directory found when no MySQL server package is installed

A data directory '/var/lib/mysql' is present on this system when no MySQL server package
is currently installed on the
system. The directory may be under control of server package received from third-party
vendors. It may also be an
```

unclaimed data directory from previous removal of mysql packages.

It is highly recommended to take data backup. If you have not done so, now would be the time to take backup in another shell. Once completed, press 'Ok' to continue.

Please provide a strong password that will be set for the root account of your MySQL database. Leave it blank to enable

password less login using UNIX socket based authentication.

Enter root password:

Now that you have selected a password for the root account, please confirm by typing it again. Do not share the password

with anyone.

Re-enter root password:

update-alternatives: using /etc/mysql/mysql.cnf to provide /etc/mysql/my.cnf (my.cnf) in auto mode

invoke-rc.d: policy-rc.d denied execution of start.

Setting up mysql-server (5.7.15-1debian8) ...

Processing triggers for libc-bin (2.19-18+deb8u4) ...

Processing triggers for systemd (215-17+deb8u4) ...

安装 MySQL 时，安装程序会提示用户输入数据库根用户的密码，并提示再次输入密码。

MySQL 程序安装完成之后，就可以通过 `mysqld` 程序启动 MySQL 的服务程序。出于安全方面的考虑，MySQL 默认禁止了以根用户运行，所以直接运行 `mysqld` 会得到错误提示。

```
root@99ac0c36e01c:/# mysqld
2016-09-24T14:03:12.712411Z 0 [Warning] TIMESTAMP with implicit DEFAULT value is deprecated. Please use --explicit_defaults_for_timestamp server option (see documentation for more details).
2016-09-24T14:03:12.713053Z 0 [Note] mysqld (mysqld 5.7.15) starting as process 463 ...
2016-09-24T14:03:12.714194Z 0 [ERROR] Fatal error: Please read "Security" section of
```

```
the manual to find out how to run mysqld as root!
```

```
2016-09-24T14:03:12.714223Z 0 [ERROR] Aborting
```

```
2016-09-24T14:03:12.714240Z 0 [Note] Binlog end
```

```
2016-09-24T14:03:12.714535Z 0 [Note] mysqld: Shutdown complete
```

因此，在运行 `mysqld` 程序之前，还需要建立一个运行 MySQL 的用户。

```
root@d71816807e72:/# groupadd -r mysql
root@d71816807e72:/# useradd -r -g mysql mysql
```

之后就可以通过该用户运行 `mysqld` 程序了。

```
root@d71816807e72:/# mysqld --user mysql
```

运行 `mysqld` 程序之后，我们发现终端出现了等待的情况，这就表明 MySQL 服务程序正在运行，等待处理来自客户端的请求。

### 8.1.3 构建 MySQL 镜像

根据前面在 Docker 容器中安装 MySQL 的过程，将构建 MySQL 镜像的 Dockerfile 写出来。

```
# MySQL
# VERSION 0.0.1

# 基础镜像
FROM debian:jessie

# 维护者信息
MAINTAINER You Ming <youming@funcuter.org>

# 增加运行用户
RUN groupadd -r mysql && useradd -r -g mysql mysql

# 增加安装源
RUN echo "deb http://repo.mysql.com/apt/debian/ jessie mysql-5.7" > /etc/apt/sources.
list.d/mysql.list \
    && apt-key adv --keyserver ha.pool.sks-keyservers.net --recv-keys A4A9406876FCBD
3C456770C88C718D3B5072E1F5 \
    && apt-get update \
    && apt-get install -y --no-install-recommends perl pwgen
```



```
# 安装 MySQL
RUN { \
# 设置 MySQL 用户密码
    echo mysql-community-server mysql-community-server/root-pass password ''; \
    echo mysql-community-server mysql-community-server/re-root-pass password ''; \
} | debconf-set-selections \
&& apt-get install -y mysql-server="5.7.15-1debian8" \
&& mkdir -p /var/lib/mysql /var/run/mysqld \
    && chown -R mysql:mysql /var/lib/mysql /var/run/mysqld \
    && chmod 777 /var/run/mysqld

# 以数据卷的形式挂载 MySQL 存储数据的目录
VOLUME /var/lib/mysql

# 暴露 MySQL 的默认端口
EXPOSE 3306

# 启动 MySQL
CMD ["mysqld", "--user", "mysql"]
```

将编写的 Dockerfile 保存到文件中，再通过 `docker build` 命令对 Dockerfile 进行构建，生成定制的 MySQL 镜像。

```
$ sudo docker build -t ymdot/mysql:0.0.1 ./mysql
Sending build context to Docker daemon 3.072 kB
Step 1 : FROM debian:jessie
----> 1b01529cc499
Step 2 : MAINTAINER You Ming <youming@funcuter.org>
----> Running in 12f69b795ele
----> 20067259f77c
Removing intermediate container 12f69b795ele
Step 3 : RUN groupadd -r mysql && useradd -r -g mysql mysql
----> Running in d50e2da57d49
----> 5e743b8ba4aa
Removing intermediate container d50e2da57d49
Step 4 : RUN echo "deb http://repo.mysql.com/apt/debian/ jessie mysql-5.7" > /etc/apt/
sources.list.d/mysql.list && apt-key adv --keyserver ha.pool.sks-keyservers.net
--recv-keys A4A9406876FCBD3C456770C88C718D3B5072E1F5 && apt-get update &&
apt-get install -y --no-install-recommends perl pwgen
----> Running in 231deb68cbdb
...
----> 2300d753fda6
Removing intermediate container 231deb68cbdb
```

```
...
---> 292099af5bf9
Removing intermediate container 62a565bc6d7f
Step 6 : VOLUME /var/lib/mysql
---> Running in ce7243e9956d
---> abfab63a22d3
Removing intermediate container ce7243e9956d
Step 7 : EXPOSE 3306
---> Running in 0ea4e38f1bd3
---> e889a91da86e
Removing intermediate container 0ea4e38f1bd3
Step 8 : CMD mysqld --user mysql
---> Running in 8835db1bc7f8
---> edd6c6ffc3df
Removing intermediate container 8835db1bc7f8
Successfully built edd6c6ffc3df
```

镜像构建完成后，可以在本地的镜像仓库中找到刚刚构建的 MySQL 镜像。

```
$ sudo docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
ymdot/mysql	0.0.1	edd6c6ffc3df	8 hours ago	517.7 MB

```
...
```

## 8.1.4 测试 MySQL 容器

通过 APT 安装 MySQL 镜像时，不但会安装 `mysqld` 服务器程序，也会安装一个名为 `mysql` 的客户端程序。下面我们使用 MySQL 客户端程序，对刚刚构建的 MySQL 镜像及其中的 MySQL 服务进行测试。

首先，基于刚刚构建的 MySQL 镜像创建一个新的容器。

```
$ sudo docker rrn -d --name mysql -p 3306:3306 ymdot/mysql:0.0.1
9a80dfcb092a5a4e1b688d6f5953b92f1868dfa5e811b0080f7c75d83ba7a24e
```

之后，创建另外一个基于 MySQL 镜像的容器，连接到刚刚创建的容器中，并进入这个新的容器。我们创建这个容器是为了使用 MySQL 镜像中的 MySQL 程序。由于 Docker 的特性，容器与容器之间完全独立，我们可以理解为这两个容器之间的连接，是服务器主机与客户端之间进行的连接。

```
$ sudo docker run -it --rm --name mysql-client --link=mysql ymdot/mysql:0.0.1 /bin/bash
root@be04b51e1b55:/#
```

将作为客户端的容器，使用 MySQL 程序连接到运行在另外一个容器中的 mysqld 程序上。

```
root@be04b51e1b55:/# mysql -h mysql -u root
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 4
Server version: 5.7.15 MySQL Community Server (GPL)

Copyright (c) 2000, 2016, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

当终端打印出 MySQL 提供的欢迎语时，表示其已经成功连接到了 mysqld 程序，下面通过几个简单的命令来查看 MySQL 程序的数据。

```
mysql> use mysql
Database changed
mysql> show tables;
+-----+
| Tables_in_mysql |
+-----+
| columns_priv |
| db |
| engine_cost |
| event |
| func |
| general_log |
| gtid_executed |
| help_category |
| help_keyword |
| help_relation |
| help_topic |
| innodb_index_stats |
| innodb_table_stats |
| ndb_binlog_index |
| plugin |
| proc |
```



```
| procs_priv          |
| proxies_priv       |
| server_cost        |
| servers            |
| slave_master_info   |
| slave_relay_log_info |
| slave_worker_info   |
| slow_log           |
| tables_priv        |
| time_zone          |
| time_zone_leap_second |
| time_zone_name     |
| time_zone_transition |
| time_zone_transition_type |
| user              |
+-----+
31 rows in set (0.00 sec)

mysql>
```

测试表明，我们构建的 MySQL 镜像是可以使用的。

## 8.2 MongoDB

MongoDB 是目前最受欢迎的非关系型数据库，在很多需要较大数据量存储支持的场景下得到了广泛应用。

### 8.2.1 MongoDB 简介

MongoDB 是面向文档的存储数据库，Mongo 一词来源于英文单词 Humongous，译为“庞大”。能够存储更多量级的数据，是 MongoDB 的一大优势。MongoDB 使用 C 语言编写，对各大系统平台都提供了支持。

MongoDB 与绝大多数非关系型数据库一样，支持松散数据的存储。它与 JSON 类似，都采用 BSON 这种结构，但以二进制的数据结构来存放数据。并且，MongoDB 还支持内存映射文件，这就使得它的存取效率在一定程度上高于单纯将数据存放于硬盘中的数据库。图 8-2 展示了 MongoDB 的标识。

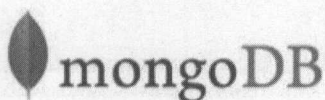


图 8-2 MongoDB

MongoDB 有强大、丰富的查询和操作方法，这让很多关系型数据库都望尘莫及。松散的数据结构配合丰富的查询功能，使 MongoDB 既能应付快速的需求及数据库设计迭代，也能方便在开发过程中轻松连接 MongoDB 进行数据的查询。可以说，MongoDB 结合了关系型数据库和非关系型数据库的优势。

在编程语言支持上，MongoDB 提供了 C、C++、Java、PHP、Python、JavaScript 等非常多的基础驱动支持。而在部署和分布形式上，MongoDB 也支持主从及副本集等形式。得益于 MongoDB 的开发维护者们，很早就成立了 MongoDB Inc.，以企业化运作的方式开发、维护和推广 MongoDB。所以，MongoDB 在实际生产中的使用率越来越高，很有可能成为新一代数据库中的佼佼者。

## 8.2.2 安装 MongoDB

MongoDB 的发展速度远远快于其他同类数据库，在系统支持上，它能够运行在所有主流的系统上，这都得益于其企业化运作模式，这一点与 Docker 的模式非常相似。

MongoDB 有社区版和企业版，我们使用免费的社区版进行安装。安装 MongoDB 的系统镜像选择使用 Ubuntu，安装方式以较为方便的 APT 安装为例。首先，创建一个基于 Ubuntu 系统镜像的容器，并进入到容器之中。

```
$ sudo docker run -it --name mongoddb ubuntu:16.04 /bin/bash
root@8b868afed3aa:/#
```

因为要通过 APT 安装 MongoDB，所以先将用于验证安装包的签名公钥加入到 APT 仓库的配置中。

```
root@8b868afed3aa:/# apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv
EA312927
Executing: /tmp/tmp.skv34CC960/gpg.1.sh --keyserver
hkp://keyserver.ubuntu.com:80
--recv
EA312927
gpg: requesting key EA312927 from hkp server keyserver.ubuntu.com
gpg: key EA312927: public key "MongoDB 3.2 Release Signing Key <packaging@mongodb.com>"
imported
```

```
gpg: Total number processed: 1
gpg:             imported: 1 (RSA: 1)
```

接着，把 MongoDB 的安装源信息加入到 APT 软件仓库中，以便下一步通过 APT 安装 MongoDB 的工作。

```
root@8b868afed3aa:/# echo "deb http://repo.mongodb.org/apt/ubuntu xenial/mongodb-org/3.2 multiverse" > /etc/apt/sources.list.d/mongodb-org-3.2.list
```

加入安装源后，更新 APT 软件仓库中的软件信息，从网络中获得详细的软件安装配置。

```
root@8b868afed3aa:/# apt-get update
Ign:1 http://repo.mongodb.org/apt/ubuntu xenial/mongodb-org/3.2 InRelease
Get:2 http://repo.mongodb.org/apt/ubuntu xenial/mongodb-org/3.2 Release [3462 B]
Get:3 http://repo.mongodb.org/apt/ubuntu xenial/mongodb-org/3.2 Release.gpg [801 B]
Get:4 http://repo.mongodb.org/apt/ubuntu xenial/mongodb-org/3.2/multiverse amd64 Packages [3481 B]
Get:5 http://archive.ubuntu.com/ubuntu xenial InRelease [247 kB]
Get:6 http://archive.ubuntu.com/ubuntu xenial-updates InRelease [95.7 kB]
Get:7 http://archive.ubuntu.com/ubuntu xenial-security InRelease [94.5 kB]
Get:8 http://archive.ubuntu.com/ubuntu xenial/main Sources [1103 kB]
Get:9 http://archive.ubuntu.com/ubuntu xenial/restricted Sources [5179 B]
Get:10 http://archive.ubuntu.com/ubuntu xenial/universe Sources [9802 kB]
Get:11 http://archive.ubuntu.com/ubuntu xenial/main amd64 Packages [1558 kB]
Get:12 http://archive.ubuntu.com/ubuntu xenial/restricted amd64 Packages [14.1 kB]
Get:13 http://archive.ubuntu.com/ubuntu xenial/universe amd64 Packages [9827 kB]
Get:14 http://archive.ubuntu.com/ubuntu xenial-updates/main Sources [240 kB]
Get:15 http://archive.ubuntu.com/ubuntu xenial-updates/universe Sources [120 kB]
Get:16 http://archive.ubuntu.com/ubuntu xenial-updates/main amd64 Packages [497 kB]
Get:17 http://archive.ubuntu.com/ubuntu xenial-updates/universe amd64 Packages [411 kB]
Get:18 http://archive.ubuntu.com/ubuntu xenial-security/main Sources [48.3 kB]
Get:19 http://archive.ubuntu.com/ubuntu xenial-security/universe Sources [10.5 kB]
Get:20 http://archive.ubuntu.com/ubuntu xenial-security/main amd64 Packages [177 kB]
Get:21 http://archive.ubuntu.com/ubuntu xenial-security/universe amd64 Packages [49.5 kB]
Fetched 24.1 MB in 20min 42s (19.4 kB/s)
Reading package lists... Done
```

然后通过 APT 软件仓库安装 MongoDB。在我们设置的软件源中，MongoDB 的名称为 `mongodb-org`，在通过 APT 安装时要使用这个名称。

```
root@8b868afed3aa:/# apt-get install -y mongodb-org
Reading package lists... Done
Building dependency tree
```



```

Reading state information... Done
The following additional packages will be installed:
  libssl1.0.0 mongodb-org-shell mongodb-org-tools
The following NEW packages will be installed:
  libssl1.0.0 mongodb-org mongodb-org-mongos mongodb-org-server mongodb-org-shell
mongodb-org-tools
0 upgraded, 6 newly installed, 0 to remove and 0 not upgraded.
Need to get 5252 kB/54.0 MB of archives.
After this operation, 239 MB of additional disk space will be used.
Get:1 http://repo.mongodb.org/apt/ubuntu xenial/mongodb-org/3.2/multiverse amd64
mongodb-org-shell amd64 3.2.9 [5252 kB]

Fetched 5252 kB in 1min 8s (76.6 kB/s)
debconf: delaying package configuration, since apt-utils is not installed
Selecting previously unselected package libssl1.0.0:amd64.
(Reading database ... 7256 files and directories currently installed.)
Preparing to unpack .../libssl1.0.0_1.0.2g-lubuntu4.5_amd64.deb ...
Unpacking libssl1.0.0:amd64 (1.0.2g-lubuntu4.5) ...
Selecting previously unselected package mongodb-org-shell.
Preparing to unpack .../mongodb-org-shell_3.2.9_amd64.deb ...
Unpacking mongodb-org-shell (3.2.9) ...
Selecting previously unselected package mongodb-org-server.
Preparing to unpack .../mongodb-org-server_3.2.9_amd64.deb ...
Unpacking mongodb-org-server (3.2.9) ...
Selecting previously unselected package mongodb-org-mongos.
Preparing to unpack .../mongodb-org-mongos_3.2.9_amd64.deb ...
Unpacking mongodb-org-mongos (3.2.9) ...
Selecting previously unselected package mongodb-org-tools.
Preparing to unpack .../mongodb-org-tools_3.2.9_amd64.deb ...
Unpacking mongodb-org-tools (3.2.9) ...
Selecting previously unselected package mongodb-org.
Preparing to unpack .../mongodb-org_3.2.9_amd64.deb ...
Unpacking mongodb-org (3.2.9) ...
Processing triggers for libc-bin (2.23-0ubuntu3) ...
Setting up libssl1.0.0:amd64 (1.0.2g-lubuntu4.5) ...
debconf: unable to initialize frontend: Dialog
debconf: (No usable dialog-like program is installed, so the dialog based frontend
cannot be used. at /usr/share/perl5/Debconf/FrontEnd/Dialog.pm line 76.)
debconf: falling back to frontend: Readline
debconf: unable to initialize frontend: Readline
debconf: (Can't locate Term/ReadLine.pm in @INC (you may need to install the
Term::ReadLine module) (@INC contains: /etc/perl /usr/local/lib/x86_64-linux-gnu/

```

```
perl/5.22.1 /usr/local/share/perl/5.22.1 /usr/lib/x86_64-linux-gnu/perl5/5.22 /usr/
share/perl5 /usr/lib/x86_64-linux-gnu/perl/5.22 /usr/share/perl/5.22 /usr/local/
lib/site_perl /usr/lib/x86_64-linux-gnu/perl-base .) at /usr/share/perl5/Debconf/
FrontEnd/Readline.pm line 7.)
debconf: falling back to frontend: Teletype
Setting up mongodb-org-shell (3.2.9) ...
Setting up mongodb-org-server (3.2.9) ...
Adding system user `mongodb' (UID 105) ...
Adding new user `mongodb' (UID 105) with group `nogroup' ...
Not creating home directory `/home/mongodb'.
Adding group `mongodb' (GID 106) ...
Done.
Adding user `mongodb' to group `mongodb' ...
Adding user mongodb to group mongodb
Done.
Setting up mongodb-org-mongos (3.2.9) ...
Setting up mongodb-org-tools (3.2.9) ...
Setting up mongodb-org (3.2.9) ...
Processing triggers for libc-bin (2.23-0ubuntu3) ...
```

安装完 MongoDB 之后, 不要马上启动程序。因为 MongoDB 的服务程序不会自动创建数据存储目录, 所以还需要先创建这个目录。

```
root@8b868afed3aa:/# mkdir -p /data/db
```

目录创建完成之后, 就可以运行 MongoDB 的服务端程序了。在刚刚安装的 MongoDB 软件包中, MongoDB 的服务端程序以 mongod 的名字出现。所以, 我们通过 mongod 程序来启动 MongoDB 的服务器程序。

```
root@8b868afed3aa:/# mongod
2016-09-24T19:13:52.267+0000 I CONTROL [initandlisten] MongoDB starting : pid=427
port=27017 dbpath=/data/db 64-bit host=8b868afed3aa
2016-09-24T19:13:52.267+0000 I CONTROL [initandlisten] db version v3.2.9
2016-09-24T19:13:52.267+0000 I CONTROL [initandlisten] git version: 22ec9e93b40c
85fc7cae7d56e7d6a02fd811088c
2016-09-24T19:13:52.267+0000 I CONTROL [initandlisten] OpenSSL version: OpenSSL
1.0.2g 1 Mar 2016
2016-09-24T19:13:52.267+0000 I CONTROL [initandlisten] allocator: tcmalloc
2016-09-24T19:13:52.267+0000 I CONTROL [initandlisten] modules: none
2016-09-24T19:13:52.267+0000 I CONTROL [initandlisten] build environment:
2016-09-24T19:13:52.267+0000 I CONTROL [initandlisten] distmod: ubuntu1604
2016-09-24T19:13:52.267+0000 I CONTROL [initandlisten] distarch: x86_64
2016-09-24T19:13:52.267+0000 I CONTROL [initandlisten] target_arch: x86_64
```



```

2016-09-24T19:13:52.267+0000 I CONTROL [initandlisten] options: {}
2016-09-24T19:13:52.270+0000 I STORAGE [initandlisten] wiredtiger_open config:
create,cache_size=1G,session_max=20000,eviction=(threads_max=4),config_base=false,
statistics=(fast),log=(enabled=true,archive=true,path=journal,compressor=snappy),
file_manager=(close_idle_time=100000),checkpoint=(wait=60,log_size=2GB),statistic
s_log=(wait=0),
2016-09-24T19:13:52.437+0000 I CONTROL [initandlisten] ** WARNING: You are running
this process as the root user, which is not recommended.
2016-09-24T19:13:52.438+0000 I CONTROL [initandlisten]
2016-09-24T19:13:52.438+0000 I CONTROL [initandlisten]
2016-09-24T19:13:52.438+0000 I CONTROL [initandlisten] ** WARNING: /sys/kernel/mm/
transparent_hugepage/enabled is 'always'.
2016-09-24T19:13:52.438+0000 I CONTROL [initandlisten] **          We suggest setting
it to 'never'
2016-09-24T19:13:52.438+0000 I CONTROL [initandlisten]
2016-09-24T19:13:52.438+0000 I CONTROL [initandlisten] ** WARNING: /sys/kernel/mm/
transparent_hugepage/defrag is 'always'.
2016-09-24T19:13:52.438+0000 I CONTROL [initandlisten] **          We suggest setting
it to 'never'
2016-09-24T19:13:52.438+0000 I CONTROL [initandlisten]
2016-09-24T19:13:52.438+0000 I FTDC [initandlisten] Initializing full-time
diagnostic data capture with directory '/data/db/diagnostic.data'
2016-09-24T19:13:52.438+0000 I NETWORK [HostnameCanonicalizationWorker] Starting
hostname canonicalization worker
2016-09-24T19:13:52.664+0000 I NETWORK [initandlisten] waiting for connections on
port 27017

```

当 MongoDB 的服务端程序完成初始化操作之后，输入指令的终端进入等待状态，这就表明 MongoDB 已经启动起来了。

### 8.2.3 构建 MongoDB 镜像

根据之前进行的 MongoDB 搭建的实践，将构建 MongoDB 镜像的过程写成相应的 Dockerfile。

```

# MongoDB
# VERSION 0.0.1

# 基础镜像
FROM ubuntu:16.04

```



```
# 维护者信息
MAINTAINER You Ming <youming@funcuter.org>

# 安装 MongoDB
RUN apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv EA312927 \
    && echo "deb http://repo.mongodb.org/apt/ubuntu xenial/mongodb-org/3.2 multiverse" \
    > /etc/apt/sources.list.d/mongodb-org-3.2.list \
    && apt-get update \
    && apt-get install -y mongodb-org=3.2.9 mongodb-org-server=3.2.9

# 以数据卷的形式挂载 MongoDB 存储数据的目录
VOLUME /data/db

# 暴露 MongoDB 的默认端口, 其中 28017 是 MongoDB 提供的 HTTP 界面的端口
EXPOSE 27017 28017

# 启动 MongoDB
CMD ["mongod", "--httpinterface"]
```

将编写好的 Dockerfile 保存到文件中, 通过 Docker 提供的 `docker build` 命令对 Dockerfile 进行构建。

```
$ sudo docker build -t ymdot/mongodb:0.0.1./mongodb
Sending build context to Docker daemon 2.56 kB
Step 1 : FROM ubuntu:16.04
----> f8d79ba03c00
Step 2 : MAINTAINER You Ming <youming@funcuter.org>
----> Running in 8430738ca910
----> 07074bb0c1dd
Removing intermediate container 8430738ca910
Step 3 : RUN apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv EA312927
&& echo "deb http://repo.mongodb.org/apt/ubuntu xenial/mongodb-org/3.2 multiverse"
> /etc/apt/sources.list.d/mongodb-org-3.2.list && apt-get update && apt-get
install -y mongodb-org=3.2.9 mongodb-org-server=3.2.9
----> Running in 7d818105ba03
...
----> 9e33e6372c36
Removing intermediate container 7d818105ba03
Step 4 : VOLUME /data/db
----> Running in 57748aled980
----> 47fclb0a3c30
Removing intermediate container 57748aled980
Step 5 : EXPOSE 27017 28017
```

```

---> Running in 40d4bf09eebe
---> 08da039c5386
Removing intermediate container 40d4bf09eebe
Step 6 : CMD mongod --httpinterface
---> Running in 702fd80a624b
---> 20da458d05f9
Removing intermediate container 702fd80a624b
Successfully built 20da458d05f9

```

完成构建 MongoDB 镜像的操作后，可以从本地镜像仓库中找到刚刚构建的 MongoDB 镜像。

```

$ sudo docker images

```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
ymdot/mongodb	0.0.1	20da458d05f9	4 minutes ago	405.5 MB
...				

## 8.2.4 测试 MongoDB 容器

在进行 MongoDB 镜像及容器的测试之前，先根据之前构建的 MongoDB 镜像建立一个新的运行 MongoDB 服务的容器。

```

$ sudo docker run -d --name mongodb -p 27017:27017 -p 28017:28017 ymdot/mongodb:0.0.1
70aeb0deb8af9271b64ec060c529bb7587d37f3b319009739a17023a5e01f95b

```

对于这个新的容器，我们开放了 27017 端口作为 MongoDB 的默认端口，也开放了 28017 端口作为 MongoDB 的 HTTP 界面的端口。

为了方便测试，这里使用 Robomongo 连接 MongoDB。Robomongo 是一个图像界面的 MongoDB 客户端，几乎提供了对 MongoDB 的全部操作，对管理 MongoDB 中的数据非常有帮助。另外，Robomongo 的界面在众多图形化的 MongoDB 客户端中是最友好的。而其跨平台性也能让用户轻松地在 Windows 和 Mac OS 中使用它进行 MongoDB 的操作。

如图 8-3 所示，打开 Robomongo，从连接列表中创建一个新的连接。在新连接配置中输入刚才创建的 MongoDB 容器的信息，并在保存后进行连接。因为我们没有修改 MongoDB 的配置，在默认的情况下，MongoDB 是不需要密码和鉴权就能进入的。

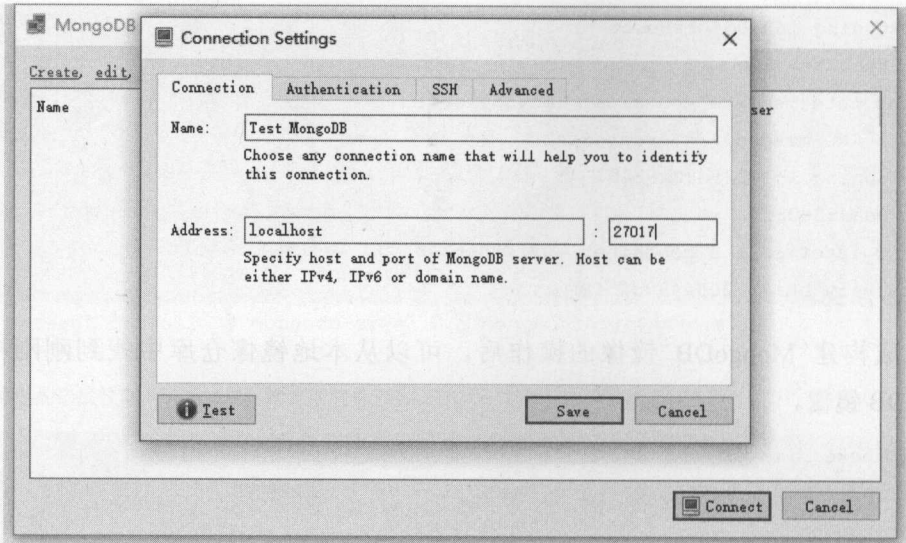


图 8-3 新连接配置

如图 8-4 所示，当通过 Robomongo 连接到 MongoDB 时，可以看到 MongoDB 中存储的数据库和数据库中的集合（Collection）列表。通过 MongoDB 的查询和其他操作命令可以在 Robomongo 中对 MongoDB 进行操作。

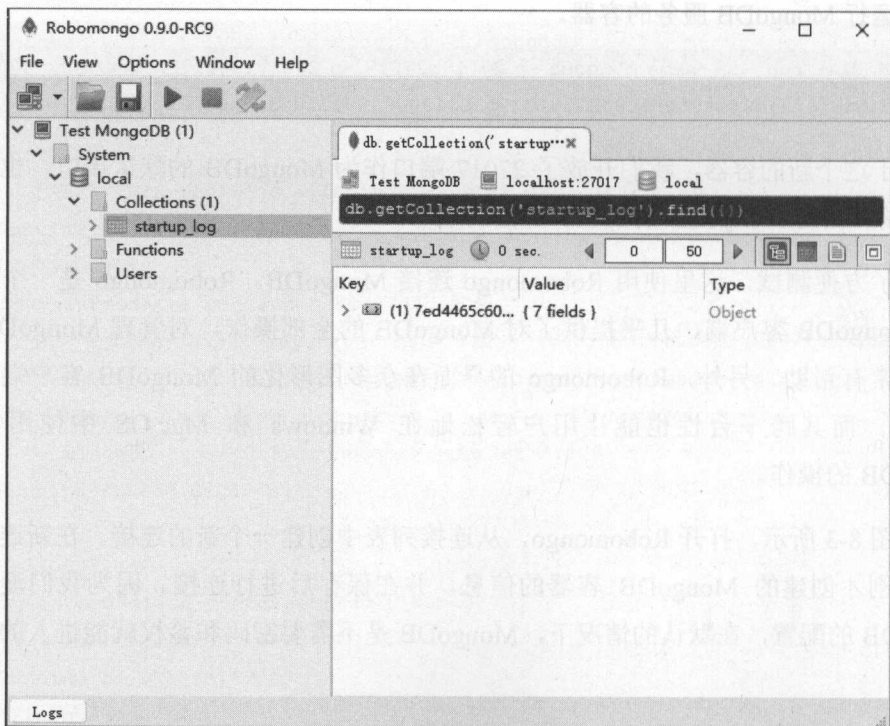


图 8-4 MongoDB 集合列表



除了使用 Robomongo，还可以通过 MongoDB 本身提供的 HTTP 管理界面对 MongoDB 进行操作。要开启 MongoDB 的 HTTP 界面，需要在启动 mongod 时带入 `--httpinterface` 参数，或者在配置文件中将其设置为开启。而且，HTTP 管理界面的默认端口为 28017，所以需要映射此端口到宿主主机上。

通过浏览器打开 MongoDB 的 HTTP 管理界面，如图 8-5 所示。

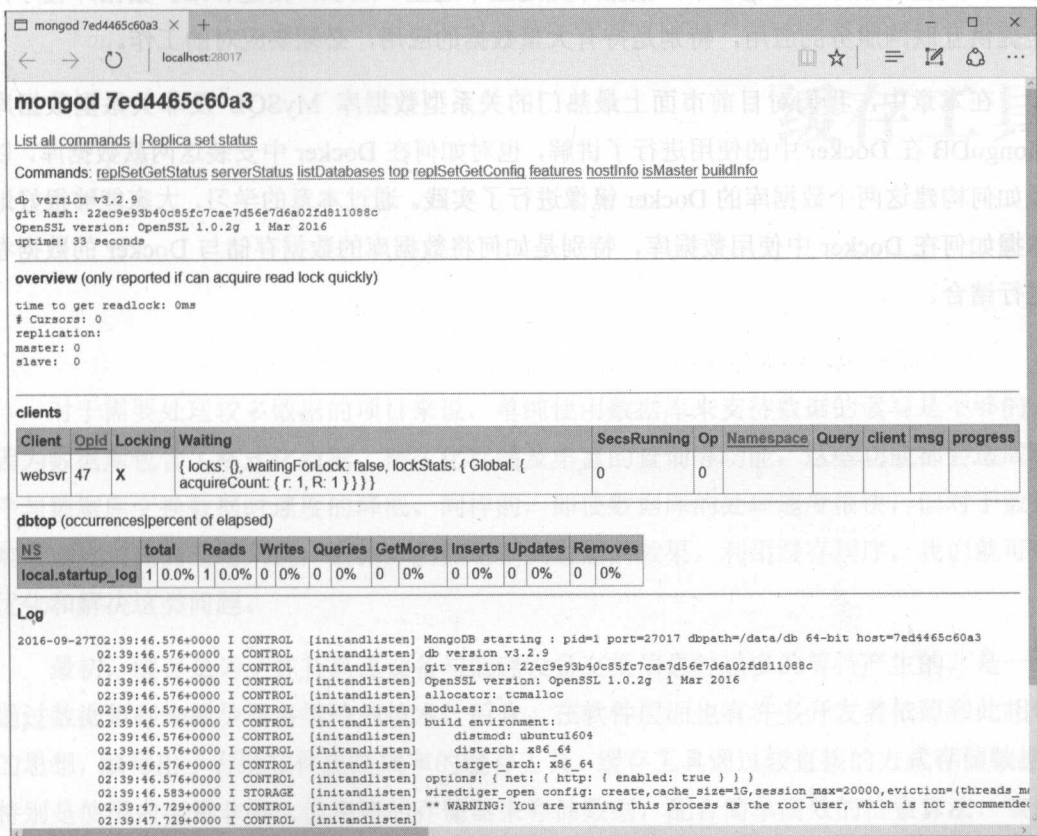


图 8-5 通过浏览器打开 MongoDB 的 HTTP 管理界面

在 MongoDB 提供的 HTTP 接口中，也能够进行大部分对 MongoDB 数据库的操作。同时，利用 HTTP 接口也能很好地监控 MongoDB 数据库的运行状态、查看运行日志等。

我们通过 Robomongo 和 MongoDB 的 HTTP 管理界面，成功连接了 MongoDB 数据库，并对 MongoDB 进行包括增、删、改、查在内的数据库基本操作。这也就说明，我们通过 Dockerfile 构建的 MongoDB 镜像是可以使用的，可以轻松通过 Dockerfile 向运行在其他主机中的 Docker 传递构建的 MongoDB 镜像。

## 8.3 本章小结

数据库是存储和管理数据的重要工具，也是任何需要处理数据的程序必备的组成部分。在大型的服务应用程序中，数据库更是重中之重。所以，搭建和维护数据库程序，是提供互联网服务的应用，特别是持有大量数据的应用，必须要应对的工作。

在本章中，我们对目前市面上最热门的关系型数据库 MySQL 及非关系型数据库 MongoDB 在 Docker 中的使用进行了讲解，也对如何在 Docker 中安装这两款数据库，以及如何构建这两个数据库的 Docker 镜像进行了实践。通过本章的学习，大家能够很好地掌握如何在 Docker 中使用数据库，特别是如何将数据库的数据存储与 Docker 的数据卷进行结合。

# 第 9 章

## 缓存工具

对于需要处理较多数据的项目来说，单纯使用数据库来支持数据的读写是不够的。因为数据库包含了格式化数据、持久化存储及丰富的查询等功能，这些功能都会造成程序与数据库交换数据时速度的降低。同样的，即使数据库的处理速度很快，但对于数据响应速度较高的应用来说，数据库仍然达不到理想的效果。利用缓存程序，我们就可以优化和解决这类问题。

最初，缓存是为了弥补高速设备与低速设备交换信息时过多的等待产生的，是一种通过数据暂存来减少设备等待的技术。后来，在软件层面也有许多开发者依照和此相似的思想，编写出了有助软件提高效率的缓存工具。缓存工具通过较直接的方式存储数据，特别是使用内存等非持久但高速的存储器来存储数据，配合简单高效的检索算法，实现了程序对数据读取的高效性。

当然，缓存工具也有一定的局限性，例如只能通过较简单的方式进行查询，以及内存的可用空间远小于硬盘等。所以在实际的架构设计中，通常把数据库与缓存工具联合起来使用，通过数据库完成复杂业务逻辑中的数据处理部分，利用缓存应对高并发和重复数据的读写。

本章就介绍在目前的服务器程序架构中，最常用的两款缓存软件，了解如何将它们部署在 Docker 中。



## 9.1 Memcached

Memcached 是一个将数据缓存在内存的工具，由于其优化合理，被广泛使用在了 Web 服务器中。

### 9.1.1 Memcached 简介

Memcached 是一个高速的缓存管理程序，遵循 BSD 开源协议，被应用在众多的服务器程序，特别是 Web 服务程序体系中。Memcached 将所有的缓存数据存放于内存之中，并结合内存优化，达到了非常高的数据读写速度。Memcached 拥有可以定义缓存过期时间的功能，也会在缓存空间已满时，根据 LRU 规则自动清理很久没有被用到的缓存项。另外，Memcached 只有键值对概念，让其可以通过简单的缓存名算法，配合各种数据库使用。图 9-1 展示了 Memcached 的标识。

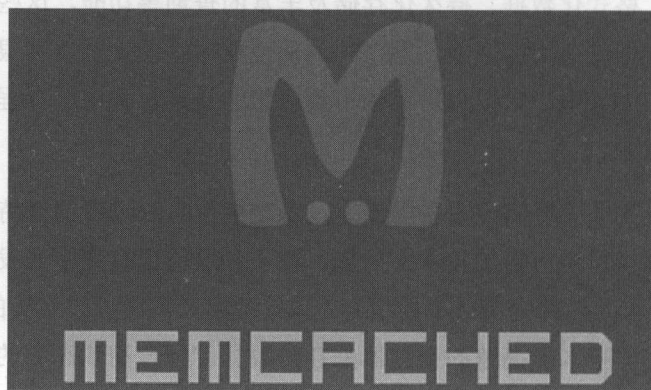


图 9-1 Memcached 的标识

Memcached 本身不具备鉴权和安全管理机制，以减少不必要的速度损失，但这也带来了一定的安全问题。为了防止 Memcached 中的数据泄露，我们通常将它放置在主机防火墙之后，只允许主机内部的程序对它进行访问。

如图 9-2 所示，对于对缓存依赖较重的应用场景，Memcached 还能进行集群扩展，用来应对更重的负载及承担容灾任务。在实现集群时，不同的 Memcached 程序之间其实是没有通信的，这样减少了网络沟通对缓存性能本身的影响。而分布式部署是根据请求 Memcached 服务的客户端和特定的算法，将不同的数据分散存储到不同的 Memcached

程序中。因为缓存本身不追求数据的可达性，所以这样的实现是在效率和需求之间最佳的实现。

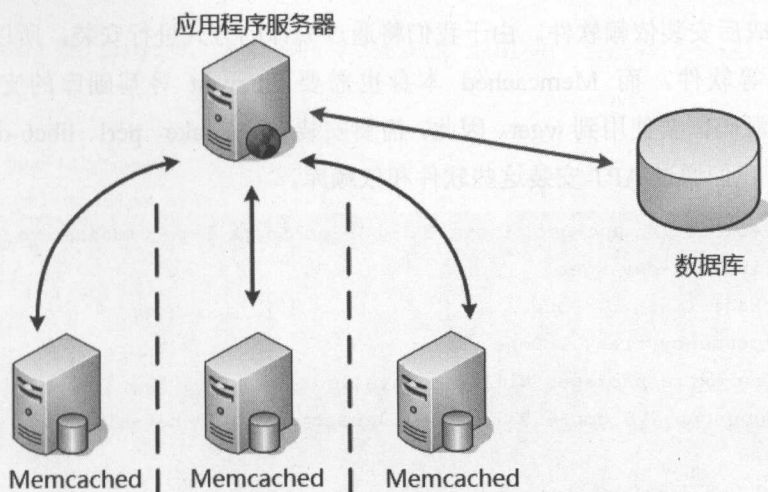


图 9-2 Memcached 集群

因此，使用 Docker 来部署 Memcached 是非常自然的。因为 Docker 与生俱来的快速部署和安全隔离性，不但能够快速完成 Memcached 集群的部署，还能够从一定程度上完成部分以往我们需要对 Memcached 进行的安全防护工作。

## 9.1.2 安装 Memcached

我们选择 Debian 系统镜像作为搭建 Memcached 服务的基础镜像。首先，在 Docker 中新建一个基于 Debian 的容器，并进入容器中。

```
$ sudo docker run --name memcached -it debian:jessie /bin/bash
root@1daaa7ea2e35:/#
```

由于需要使用 APT 软件仓库安装部分 Memcached 所需的依赖软件和连接库，所以要先对 APT 软件仓库进行更新，以保障软件仓库中的软件信息是最新的。

```
root@1daaa7ea2e35:/# apt-get update
Ign http://httpredir.debian.org jessie InRelease
Get:1 http://httpredir.debian.org jessie-updates InRelease [142 kB]
Get:2 http://httpredir.debian.org jessie Release.gpg [2373 B]
Get:3 http://httpredir.debian.org jessie Release [148 kB]
Get:4 http://httpredir.debian.org jessie-updates/main amd64 Packages [17.6 kB]
Get:5 http://httpredir.debian.org jessie/main amd64 Packages [9064 kB]
Get:6 http://security.debian.org jessie/updates InRelease [63.1 kB]
```

```
Get:7 http://security.debian.org jessie/updates/main amd64 Packages [389 kB]
Fetched 9827 kB in 1min 43s (94.7 kB/s)
Reading package lists... Done
```

更新完成后安装依赖软件。由于我们将通过编译的方式进行安装，所以需要用到 gcc、make 等软件，而 Memcached 本身也需要 libevent 等基础库的支持，且下载 Memcached 源码需要使用到 wget。因此，需要安装 gcc、make、perl、libc6-dev、libevent-dev、wget。我们通过 APT 安装这些软件和依赖库。

```
root@1daaa7ea2e35:/# apt-get install -y --no-install-recommends gcc make perl
libc6-dev libevent-dev wget
Reading package lists... Done
Building dependency tree... Done
The following extra packages will be installed:
  binutils cpp cpp-4.9 gcc-4.9 libasan1 libatomic1 libc-dev-bin libc6 libcilkrts5
  libcloog-isl4
  libevent-2.0-5 libevent-core-2.0-5 libevent-extra-2.0-5 libevent-openssl-2.0-5
  libevent-pthreads-2.0-5 libffi6 libgcc-4.9-dev libgdbm3 libgmp10 libgnutls-deb0-28
  libgomp1
  libhogweed2 libicu52 libidn11 libisl10 libitm1 liblsan0 libmpc3 libmpfr4 libnettle4
  libp11-kit0
  libpsl0 libquadmath0 libssl1.0.0 libtasn1-6 libtsan0 libubsan0 linux-libc-dev
  perl-base perl-modules
Suggested packages:
  binutils-doc cpp-doc gcc-4.9-locales gcc-multilib manpages-dev autoconf automake
  libtool flex bison
  gdb gcc-doc gcc-4.9-multilib gcc-4.9-doc libgcc1-dbg libgomp1-dbg libitm1-
  dbg libatomic1-dbg
  libasan1-dbg liblsan0-dbg libtsan0-dbg libubsan0-dbg libcilkrts5-dbg libquadmath0-
  dbg glibc-doc
  gnutls-bin make-doc perl-doc libterm-readline-gnu-perl libterm-readline-perl-
  perl libb-lint-perl
  libcpanplus-dist-build-perl libcpanplus-perl libfile-checktree-perl liblog-
  message-simple-perl
  liblog-message-perl libobject-accessor-perl
Recommended packages:
  rename libarchive-extract-perl libmodule-pluggable-perl libpod-latex-perl libterm-
  ui-perl
  libtext-soundex-perl libcgi-pm-perl libmodule-build-perl libpackage-constants-
  perl ca-certificates
The following NEW packages will be installed:
  binutils cpp cpp-4.9 gcc gcc-4.9 libasan1 libatomic1 libc-dev-bin libc6-dev
```



```

libcilkrts5
  libcloop-is14 libevent-2.0-5 libevent-core-2.0-5 libevent-dev libevent-extra-2.0-5
  libevent-openssl-2.0-5 libevent-pthreads-2.0-5 libffi6 libgcc-4.9-dev libgdbm3
libgmp10
  libgnutls-deb0-28 libgomp1 libhogweed2 libicu52 libidn11 libisl10 libitm1 liblsan0
libmpc3 libmpfr4
  libnettle4 libp11-kit0 libpsl0 libquadmath0 libssl1.0.0 libtasn1-6 libtsan0
libubsan0 linux-libc-dev
  make perl perl-modules wget
The following packages will be upgraded:
  libc6 perl-base
2 upgraded, 44 newly installed, 0 to remove and 16 not upgraded.
Need to get 43.3 MB of archives.
After this operation, 162 MB of additional disk space will be used.
Get:1 http://httpredir.debian.org/debian/ jessie/main perl-base amd64 5.20.2-3+
deb8u6 [1229 kB]
Get:2 http://httpredir.debian.org/debian/ jessie/main libc6 amd64 2.19-18+deb8u6
[4665 kB]
Get:3 http://httpredir.debian.org/debian/ jessie/main libgdbm3 amd64 1.8.3-13.1 [30.0 kB]
...

```

Memcached 的源码可以从 <http://memcached.org/> 中获取，目前 Memcached 的最新版为 1.4.31，源代码的下载地址为 <http://memcached.org/files/memcached-1.4.31.tar.gz>，我们以这个版本为例进行讲解。

通过 wget 将 Memcached 的源码下载到临时文件目录中，并将其解压。

```

root@l1daaa7ea2e35:/# cd /tmp
root@l1daaa7ea2e35:/tmp# wget -O memcached.tar.gz "http://memcached.org/files/
memcached-1.4.31.tar.gz"
converted 'http://memcached.org/files/memcached-1.4.31.tar.gz' (ANSI_X3.4-1968) ->
'http://memcached.org/files/memcached-1.4.31.tar.gz' (UTF-8)
--2016-09-19 05:55:02-- http://memcached.org/files/memcached-1.4.31.tar.gz
Resolving memcached.org (memcached.org)... 173.255.253.96
Connecting to memcached.org (memcached.org)|173.255.253.96|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 389502 (380K) [application/x-tar]
Saving to: 'memcached.tar.gz'

memcached.tar.gz      100%[=====>] 380.37K  101KB/s  in 3.8s

2016-09-19 05:55:19 (101 KB/s) - 'memcached.tar.gz' saved [389502/389502]

root@l1daaa7ea2e35:/tmp# tar -xzf memcached.tar.gz

```

接着，进入到下载和解压的 Memcached 源码目录，执行运行编译前的配置、编译、安装命令。

```
root@1daaa7ea2e35:/tmp# cd memcached-1.4.31/
root@1daaa7ea2e35:/tmp/memcached-1.4.31# ./configure && make && make install
checking build system type... x86_64-pc-linux-gnu
checking host system type... x86_64-pc-linux-gnu
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... /bin/mkdir -p
checking for gawk... no
checking for mawk... mawk
checking whether make sets $(MAKE)... yes
checking whether make supports nested variables... yes
checking for gcc... gcc
checking whether the C compiler works... yes
checking for C compiler default output file name... a.out
checking for suffix of executables...
checking whether we are cross compiling... no
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
...
make all-recursive
make[1]: Entering directory '/tmp/memcached-1.4.31'
Making all in doc
make[2]: Entering directory '/tmp/memcached-1.4.31/doc'
make all-am
make[3]: Entering directory '/tmp/memcached-1.4.31/doc'
make[3]: Nothing to be done for 'all-am'.
make[3]: Leaving directory '/tmp/memcached-1.4.31/doc'
make[2]: Leaving directory '/tmp/memcached-1.4.31/doc'
make[2]: Entering directory '/tmp/memcached-1.4.31'
gcc -std=gnu99 -DHAVE_CONFIG_H -I. -DNDEBUG -g -O2 -pthread -pthread -Wall -Werror
-pedantic -Wmissing-prototypes -Wmissing-declarations -Wredundant-decls -MT
memcached-memcached.o -MD -MP -MF .deps/memcached-memcached.Tpo -c -o memcached-
memcached.o `test -f 'memcached.c' || echo './'`memcached.c
mv -f .deps/memcached-memcached.Tpo .deps/memcached-memcached.Po
gcc -std=gnu99 -DHAVE_CONFIG_H -I. -DNDEBUG -g -O2 -pthread -pthread -Wall -Werror
-pedantic -Wmissing-prototypes -Wmissing-declarations -Wredundant-decls -MT
memcached-hash.o -MD -MP -MF .deps/memcached-hash.Tpo -c -o memcached-hash.o `test
-f 'hash.c' || echo './'`hash.c
```



```

mv -f .deps/memcached-hash.Tpo .deps/memcached-hash.Po
gcc -std=gnu99 -DHAVE_CONFIG_H -I. -DNDEBUG -g -O2 -pthread -pthread -Wall -Werror
-pedantic -Wmissing-prototypes -Wmissing-declarations -Wredundant-decls -MT
memcached-jenkins_hash.o -MD -MP -MF .deps/memcached-jenkins_hash.Tpo -c -o
memcached-jenkins_hash.o `test -f 'jenkins_hash.c' || echo './'`jenkins_hash.c
mv -f .deps/memcached-jenkins_hash.Tpo .deps/memcached-jenkins_hash.Po
gcc -std=gnu99 -DHAVE_CONFIG_H -I. -DNDEBUG -g -O2 -pthread -pthread -Wall -Werror
-pedantic -Wmissing-prototypes -Wmissing-declarations -Wredundant-decls -MT
memcached-murmur3_hash.o -MD -MP -MF .deps/memcached-murmur3_hash.Tpo -c -o
memcached-murmur3_hash.o `test -f 'murmur3_hash.c' || echo './'`murmur3_hash.c
mv -f .deps/memcached-murmur3_hash.Tpo .deps/memcached-murmur3_hash.Po
gcc -std=gnu99 -DHAVE_CONFIG_H -I. -DNDEBUG -g -O2 -pthread -pthread -Wall -Werror
-pedantic -Wmissing-prototypes -Wmissing-declarations -Wredundant-decls -MT
memcached-slabs.o -MD -MP -MF .deps/memcached-slabs.Tpo -c -o memcached-slabs.o `test
-f 'slabs.c' || echo './'`slabs.c
mv -f .deps/memcached-slabs.Tpo .deps/memcached-slabs.Po
gcc -std=gnu99 -DHAVE_CONFIG_H -I. -DNDEBUG -g -O2 -pthread -pthread -Wall -Werror
-pedantic -Wmissing-prototypes -Wmissing-declarations -Wredundant-decls -MT
memcached-items.o -MD -MP -MF .deps/memcached-items.Tpo -c -o memcached-items.o `test
-f 'items.c' || echo './'`items.c
mv -f .deps/memcached-items.Tpo .deps/memcached-items.Po
...
make install-recursive
make[1]: Entering directory '/tmp/memcached-1.4.31'
Making install in doc
make[2]: Entering directory '/tmp/memcached-1.4.31/doc'
make install-am
make[3]: Entering directory '/tmp/memcached-1.4.31/doc'
make[4]: Entering directory '/tmp/memcached-1.4.31/doc'
make[4]: Nothing to be done for 'install-exec-am'.
/bin/mkdir -p '/usr/local/share/man/man1'
/usr/bin/install -c -m 644 memcached.1 '/usr/local/share/man/man1'
make[4]: Leaving directory '/tmp/memcached-1.4.31/doc'
make[3]: Leaving directory '/tmp/memcached-1.4.31/doc'
make[2]: Leaving directory '/tmp/memcached-1.4.31/doc'
make[2]: Entering directory '/tmp/memcached-1.4.31'
make[3]: Entering directory '/tmp/memcached-1.4.31'
/bin/mkdir -p '/usr/local/bin'
/usr/bin/install -c memcached '/usr/local/bin'
/bin/mkdir -p '/usr/local/include/memcached'
/usr/bin/install -c -m 644 protocol_binary.h '/usr/local/include/memcached'
make[3]: Leaving directory '/tmp/memcached-1.4.31'

```



```
make[2]: Leaving directory '/tmp/memcached-1.4.31'
make[1]: Leaving directory '/tmp/memcached-1.4.31'
```

至此，Memcached 就安装完成了，我们可以尝试运行 Memcached。由于 Memcached 默认是不允许使用根用户运行的，所以直接运行 Memcached 程序会得到无法以根用户运行的提示。

```
root@1daaa7ea2e35:/tmp/memcached-1.4.31# cd /
root@1daaa7ea2e35:/# memcached
can't run as root without the -u switch
```

因此先创建一个运行 Memcached 的用户，再使用 Memcached 的 -u 参数指定 Memcached 的运行用户。

```
root@1daaa7ea2e35:/# groupadd -r memcached
root@1daaa7ea2e35:/# useradd -r -g memcached memcached
root@1daaa7ea2e35:/# memcached -u memcached
```

当输入命令的终端出现等待时，表示 Memcached 已经运行了。默认情况下，Memcached 以前台方式运行，所以不需要进行前台运行的配置。

## 9.1.3 构建 Memcached 镜像

根据我们之前的实践，可以把安装 Memcached 的过程写入到 Dockerfile 中，用来构建运行 Memcached 服务的 Docker 镜像。

```
# Memcached
# VERSION 0.0.1

# 基础镜像
FROM debian:jessie

# 维护者信息
MAINTAINER You Ming <youming@funcuter.org>

# 创建运行用户
RUN groupadd -r memcached && useradd -r -g memcached memcached

# 安装 Memcached
RUN apt-get update \
# 编译安装时记录下需要的依赖包，安装完成后移除，这样就减少了镜像层的空间
&& buildDeps='gcc make perl libc6-dev libevent-dev wget' \
&& apt-get install -y --no-install-recommends libevent-2.0-5 $buildDeps \
```

```

# 下载 Memcached 源码
&& wget -O memcached.tar.gz "http://memcached.org/files/memcached-1.4.31.tar.gz" \
&& mkdir -p /usr/src/memcached \
&& tar -xzf memcached.tar.gz -C /usr/src/memcached --strip-components=1 \
&& rm memcached.tar.gz \
&& cd /usr/src/memcached \
# 配置、编译、安装
&& ./configure \
&& make \
&& make install \
&& cd / \
&& rm -rf /usr/src/memcached \
# 删除编译使用的软件和依赖包
&& apt-get purge -y --auto-remove $buildDeps

# 使用创建的用户来运行 Memcached
USER memcached

# 暴露 Memcached 的默认端口 11211
EXPOSE 11211

# 启动 Memcached
CMD ["memcached"]

```

将编写好的 Dockerfile 保存成文件，通过 `docker build` 命令将这个 Dockerfile 构建成为 Memcached 镜像。

```

$ sudo docker build -t ymdot/memcached:0.0.1 ./memcached
Sending build context to Docker daemon 3.072 kB
Step 1 : FROM debian:jessie
---> 1b088884749b
Step 2 : MAINTAINER You Ming <youming@funcuter.org>
---> Running in 21b5698891ce
---> fc23840ad555
Removing intermediate container 21b5698891ce
Step 3 : RUN groupadd -r memcached && useradd -r -g memcached memcached
---> Running in 153821479af3
---> f73822534fec
Removing intermediate container 153821479af3
Step 4 : RUN apt-get update && buildDeps='gcc make perl libc6-dev libevent-dev wget'
&& apt-get install -y --no-install-recommends $buildDeps && wget -O memcached.
tar.gz "http://memcached.org/files/memcached-1.4.31.tar.gz" && mkdir -p /usr/
src/memcached && tar -xzf memcached.tar.gz -C /usr/src/memcached --strip-

```

```

components=1  && rm memcached.tar.gz && cd /usr/src/memcached  && ./configure
&& make  && make install  && cd /  && rm -rf /usr/src/memcached  &&
apt-get purge -y --auto-remove $buildDeps
---> Running in 6072c2f9d8c5
...
---> 2fd88e707c47
Removing intermediate container 6072c2f9d8c5
Step 5 : USER memcached
---> Running in 178cb91f8f75
---> ae12326d217c
Removing intermediate container 178cb91f8f75
Step 6 : EXPOSE 11211
---> Running in d27625314c21
---> 4eeb76f8f096
Removing intermediate container d27625314c21
Step 7 : CMD memcached
---> Running in cc39e837ceda
---> 3e371c334c04
Removing intermediate container cc39e837ceda
Successfully built 3e371c334c04

```

镜像构建完成后可以在本地镜像仓库中找到。

```

$ sudo docker images

```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
yndot/memcached	0.0.1	3e371c334c04	9 hours ago	152.7 MB
...				

## 9.1.4 测试 Memcached 容器

通过上面构建的 Memcached 镜像创建一个新的容器，并将容器中 Memcached 所监听的 11211 端口映射到物理主机上。

由于 Memcached 支持简单的指令格式，这就可以让我们通过 Telnet 等简单工具连接到 Memcached，并向它发送指令来测试 Memcached 的运行情况。

```

$ telnet 127.0.0.1 11211
...

```

我们通过 Memcached 的 add、replace 和 get 三个指令来模拟一个简单的缓存写入和读取的流程。

add、replace 和 get 三个指令的使用格式如下。



```
add <key> <flags> <exptime> <bytes>\r\n<value>\r\n
replace <Key> <flags> <exptime> <bytes>\r\n<value>\r\n
get <key>
```

其中,

- ☐ key: 缓存的名称。
- ☐ flags: 标记位, 用于判断缓存的一致性。
- ☐ exptime: 缓存的过期时间, 如果为 0 则采用 Memcached 允许的最大时间。
- ☐ bytes: 缓存数据占用的字节长度。
- ☐ value: 缓存的数据。

先通过 add 指令增加一条名为 hello、值为 world 的缓存数据。

```
add hello 1 0 5
world
STORED
```

接着通过 get 指令取得这条缓存数据的值。

```
get hello
VALUE hello 1 5
world
END
```

可以看到, 缓存确实被存放到了 Memcached 中。

再使用 replace 替换 hello 这条缓存数据的值。

```
replace hello 1 0 5
kitty
STORED
```

然后通过 get 指令查看这条缓存。

```
get hello 1 0
VALUE hello 1 5
kitty
END
```

通过 Memcached 返回的结果, 可以发现缓存的值已经被替换。

经过简单的测试, 我们是能够正常地连接和将数据缓存到运行在 Docker 容器中的 Memcached 上的, 这就表示我们构建的 Memcached 镜像是成功的。

## 9.2 Redis

Redis 是非关系型数据库，但其内存镜像的机制，使其在缓存工具的领域更受欢迎。目前已经有 many 网站将 Redis 作为主要的缓存工具。

### 9.2.1 Redis 简介

Redis 是开源的键值对存储数据库，支持多种存储类型，是目前最受欢迎的 KV 数据库。为什么将一个数据库纳入缓存工具的范围呢？因为 Redis 主要基于内存存储数据，其查询效率与 Memcached 这种缓存工具不相上下，而 Redis 所提供的关于数据库方面的功能并不多，在数据的持久化方面做得也不是特别可靠，所以我们通常不把 Redis 当作可靠的数据库使用，而是把它用在缓存上。

支持多种数据类型是 Redis 的一大优势，当把 Redis 用作缓存工具时，这种优势也能体现出不错的效果。Redis 支持的数据类型包括字符串 (String)、散列 (Hash)、列表 (List)、集合 (Set)、有序集合 (Sorted Set) 等。Redis 拥有多种数据类型的特性，使其不但能够完成本职的数据库工作，也能够胜任缓存、消息中间件等任务。图 9-3 展示了 Redis 的标识。



图 9-3 Redis 的标识

另外，Redis 还支持数据持久化、事务、Lua 脚本等功能，主从分离、数据分片等用于集群部署的功能更使 Redis 能够很好地应对分布式部署环境。目前，微博已经大范围采用 Redis 作为其业务系统抗压环节的重要组成部分。

### 9.2.2 安装 Redis

我们使用 Debian 镜像作为搭建 Redis 服务的基础镜像。首先，通过 `docker run` 命令创建一个基于 Debian 镜像的容器，并进入到容器中。

```
$ sudo docker run -it --name redis debian:jessie /bin/bash
root@69ecb9ec444d:/#
```

由于 Debian 镜像是一个精简镜像，内置的软件非常少，且缺乏编译 Redis 需要的基本工具，所以这里利用 APT 软件仓库安装编译软件和依赖库。在安装需要的编译软件和依赖库之前，先进行 APT 软件仓库的更新，并获取最新的软件列表及软件下载源等信息。

```
root@69ecb9ec444d:/# apt-get update
Ign http://httpredir.debian.org jessie InRelease
Get:1 http://httpredir.debian.org jessie-updates InRelease [142 kB]
Get:2 http://security.debian.org jessie/updates InRelease [63.1 kB]
Get:3 http://httpredir.debian.org jessie Release.gpg [2373 B]
Get:4 http://security.debian.org jessie/updates/main amd64 Packages [389 kB]
Get:5 http://httpredir.debian.org jessie Release [148 kB]
Get:6 http://httpredir.debian.org jessie-updates/main amd64 Packages [17.6 kB]
Get:7 http://httpredir.debian.org jessie/main amd64 Packages [9064 kB]
Fetched 9827 kB in 1min 18s (125 kB/s)
Reading package lists... Done
```

下载 Redis 需要使用 `wget`，编译 Redis 需要 `gcc` 和 `make` 来完成。同时，还需要 `libc6` 库的头文件，所以需要安装 APT 软件仓库中的 `libc6-dev`。我们通过 APT 安装这几个软件和依赖包。

```
root@69ecb9ec444d:/# apt-get install -y --no-install-recommends wget gcc make libc-dev
Reading package lists... Done
Building dependency tree... Done
Note, selecting 'libc6-dev' instead of 'libc-dev'
The following extra packages will be installed:
  binutils cpp cpp-4.9 gcc-4.9 libasan1 libatomic1 libc-dev-bin libc6 libcilkrts5
  libcloog-isl4
  libffi6 libgcc-4.9-dev libgmp10 libgnutls-deb0-28 libgomp1 libhogweed2 libicu52
  libidn11 libisl10
  libitm1 liblsan0 libmpc3 libmpfr4 libnettle4 libp11-kit0 libpsl0 libquadmath0
  libtasn1-6 libtsan0
  libubsan0 linux-libc-dev
Suggested packages:
  binutils-doc cpp-doc gcc-4.9-locales gcc-multilib manpages-dev autoconf automake
  libtool flex bison
  gdb gcc-doc gcc-4.9-multilib gcc-4.9-doc libgcc1-dbg libgomp1-dbg libitm1-dbg
  libatomic1-dbg
  libasan1-dbg liblsan0-dbg libtsan0-dbg libubsan0-dbg libcilkrts5-dbg libquadmath0-
  dbg glibc-doc
  gnutls-bin make-doc
```



```
Recommended packages:
  ca-certificates

The following NEW packages will be installed:
  binutils cpp cpp-4.9 gcc gcc-4.9 libasan1 libatomic1 libc-dev-bin libc6-dev
  libcilkrts5
  libcloog-isl4 libffi6 libgcc-4.9-dev libgmp10 libgnutls-deb0-28 libgomp1 libhogweed2
  libicu52
  libidn11 libisl10 libitm1 liblsan0 libmpc3 libmpfr4 libnettle4 libp11-kit0 libpsl0
  libquadmath0
  libtasn1-6 libtsan0 libubsan0 linux-libc-dev make wget

The following packages will be upgraded:
  libc6
1 upgraded, 34 newly installed, 0 to remove and 17 not upgraded.
Need to get 35.2 MB of archives.
After this operation, 123 MB of additional disk space will be used.
Get:1 http://httpredir.debian.org/debian/ jessie/main libc6 amd64 2.19-18+deb8u6
[4665 kB]
...
```

目前最新的 Redis 稳定版本是 3.2.3，可以通过网址 <http://download.redis.io/releases/redis-3.2.3.tar.gz> 下载到 3.2.3 版本的 Redis 源码包。在容器里通过 `wget` 将源码包下载到临时目录中，并进行源码包的解压。

```
root@69ecb9ec444d:/# cd /tmp
root@69ecb9ec444d:/tmp# wget -O redis.tgz http://download.redis.io/releases/redis-3.2.3.tar.gz
converted 'http://download.redis.io/releases/redis-3.2.3.tar.gz' (ANSI_X3.4-1968)
-> 'http://download.redis.io/releases/redis-3.2.3.tar.gz' (UTF-8)
--2016-09-21 04:59:22-- http://download.redis.io/releases/redis-3.2.3.tar.gz
Resolving download.redis.io (download.redis.io)... 109.74.203.151
Connecting to download.redis.io (download.redis.io)|109.74.203.151|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1541401 (1.5M) [application/x-gzip]
Saving to: 'redis.tgz'

redis.tgz          100%[=====>] 1.47M 69.2KB/s in 33s

2016-09-21 04:59:56 (45.8 KB/s) - 'redis.tgz' saved [1541401/1541401]

root@69ecb9ec444d:/tmp# tar xzf redis.tgz
```

接着进入 Redis 源码目录进行编译。Redis 是非常小巧的软件，以至于它的开发者可

以不考虑太多的环境配置，直接写出 Makefile 来使用。所以不需要使用 autoconf 来生成 Makefile，可以直接通过 make 命令来编译 Redis。

```

root@69ecb9ec444d:/tmp# cd redis-3.2.3
root@69ecb9ec444d:/tmp/redis-3.2.3# make && make install
cd src && make all
make[1]: Entering directory '/tmp/redis-3.2.3/src'
rm -rf redis-server redis-sentinel redis-cli redis-benchmark redis-check-rdb
redis-check-aof *.o *.gcda *.gcno *.gcov redis.info lcov-html
(cd ../deps && make distclean)
make[2]: Entering directory '/tmp/redis-3.2.3/deps'
(cd hiredis && make clean) > /dev/null || true
(cd linenoise && make clean) > /dev/null || true
(cd lua && make clean) > /dev/null || true
(cd geohash-int && make clean) > /dev/null || true
(cd jemalloc && [ -f Makefile ] && make distclean) > /dev/null || true
(rm -f .make-*)
make[2]: Leaving directory '/tmp/redis-3.2.3/deps'
(rm -f .make-*)
echo STD=-std=c99 -pedantic -DREDIS_STATIC='' >> .make-settings
echo WARN=-Wall -W >> .make-settings
echo OPT=-O2 >> .make-settings
echo MALLOC=jemalloc >> .make-settings
echo CFLAGS= >> .make-settings
echo LDFLAGS= >> .make-settings
echo REDIS_CFLAGS= >> .make-settings
echo REDIS_LDFLAGS= >> .make-settings
echo PREV_FINAL_CFLAGS=-std=c99 -pedantic -DREDIS_STATIC='' -Wall -W -O2 -g -ggdb
-I../deps/geohash-int -I../deps/hiredis -I../deps/linenoise -I../deps/lua/src
-DUSE_JEMALLOC -I../deps/jemalloc/include >> .make-settings
echo PREV_FINAL_LDFLAGS= -g -ggdb -rdynamic >> .make-settings
(cd ../deps && make hiredis linenoise lua geohash-int jemalloc)
make[2]: Entering directory '/tmp/redis-3.2.3/deps'
(cd hiredis && make clean) > /dev/null || true
(cd linenoise && make clean) > /dev/null || true
(cd lua && make clean) > /dev/null || true
(cd geohash-int && make clean) > /dev/null || true
(cd jemalloc && [ -f Makefile ] && make distclean) > /dev/null || true
(rm -f .make-*)
(echo "" > .make-cflags)
(echo "" > .make-ldflags)
MAKE hiredis

```



```
...
cd src && make install
make[1]: Entering directory '/tmp/redis-3.2.3/src'

Hint: It's a good idea to run 'make test' ;)

INSTALL install
INSTALL install
INSTALL install
INSTALL install
INSTALL install

make[1]: Leaving directory '/tmp/redis-3.2.3/src'
```

Redis 的服务程序名称为 redis-server，完成对 Redis 的编译和安装后可以通过这个命令来运行 Redis。

```
root@69ecb9ec444d:/tmp/redis-3.2.3# redis-server
3798:C 21 Sep 05:09:41.376 # Warning: no config file specified, using the default config.
In order to specify a config file use redis-server /path/to/redis.conf
```

```

Redis 3.2.3 (00000000/0) 64 bit
Running in standalone mode
Port: 6379
PID: 3798

http://redis.io

```

```
3798:M 21 Sep 05:09:41.378 # WARNING: The TCP backlog setting of 511 cannot be enforced
because /proc/sys/net/core/somaxconn is set to the lower value of 128.
3798:M 21 Sep 05:09:41.378 # Server started, Redis version 3.2.3
3798:M 21 Sep 05:09:41.378 # WARNING overcommit_memory is set to 0! Background save
may fail under low memory condition. To fix this issue add 'vm.overcommit_memory =
1' to /etc/sysctl.conf and then reboot or run the command 'sysctl vm.overcommit
```



```
memory=1' for this to take effect.
3798:M 21 Sep 05:09:41.378 # WARNING you have Transparent Huge Pages (THP) support
enabled in your kernel. This will create latency and memory usage issues with Redis.
To fix this issue run the command 'echo never > /sys/kernel/mm/transparent_
hugepage/enabled' as root, and add it to your /etc/rc.local in order to retain the
setting after a reboot. Redis must be restarted after THP is disabled.
3798:M 21 Sep 05:09:41.378 * The server is now ready to accept connections on port
6379
```

默认情况下, Redis 会以前台的模式运行, 启动 `redis-server` 后, 屏幕上打印出 Redis 程序运行的相关信息, 以及一个由字符绘成的 Redis 标识。当这些信息打印完成后, 终端出现等待, 这就表示 Redis 程序已经开始提供服务。

### 9.2.3 构建 Redis 镜像

结合之前安装 Redis 的实践, 可以将构建 Redis 镜像的过程写成对应的 Dockerfile, 内容如下。

```
# Redis
# VERSION 0.0.1

# 基础镜像
FROM debian:jessie

# 维护者信息
MAINTAINER You Ming <youming@funcuter.org>

# 安装 Redis
RUN apt-get update \

# 编译安装时记录下需要的依赖包, 安装完成后移除, 以减少镜像层的空间
&& buildDeps='gcc make libc6-dev wget' \
&& apt-get install -y --no-install-recommends $buildDeps \

# 下载 Redis 源码
&& wget -O redis.tgz "http://download.redis.io/releases/redis-3.2.3.tar.gz" \
&& mkdir -p /usr/src/redis \
&& tar -xzf redis.tgz -C /usr/src/redis --strip-components=1 \
&& rm redis.tgz \
&& cd /usr/src/redis \
&& make \
&& make install \
&& cd / \
&& rm -rf /usr/src/redis \
```

```
&& apt-get purge -y --auto-remove $buildDeps

# 暴露 Redis 的默认端口 6379
EXPOSE 6379

# 启动 Redis
CMD ["redis-server"]
```

将编写好的 Dockerfile 保存到文件中，再进入命令行终端，通过 `docker build` 命令对 Dockerfile 进行构建。

```
$ sudo docker build -t ymdot/redis:0.0.1 ./redis
Sending build context to Docker daemon 2.56 kB
Step 1 : FROM debian:jessie
----> 1b088884749b
Step 2 : MAINTAINER You Ming <youming@funcuter.org>
----> Running in f5e7be9e14bb
----> 0b647e0ed3ee
Removing intermediate container f5e7be9e14bb
Step 3 : RUN apt-get update && buildDeps='gcc make libc6-dev wget' && apt-get
install -y --no-install-recommends $buildDeps && wget -O redis.tgz
"http://download.redis.io/releases/redis-3.2.3.tar.gz" && mkdir -p
/usr/src/redis && tar -xzf redis.tgz -C /usr/src/redis --strip-components=1
&& rm redis.tgz && cd /usr/src/redis && make && make install
&& cd / && rm -rf /usr/src/redis && apt-get purge -y --auto-remove $buildDeps
----> Running in 2a8e831ba913
Get:1 http://security.debian.org jessie/updates InRelease [63.1 kB]
...
----> 4bd47c5be970
Removing intermediate container 2a8e831ba913
Step 4 : EXPOSE 6379
----> Running in 8c5b348ab36c
----> c80ff49a0be2
Removing intermediate container 8c5b348ab36c
Step 5 : CMD redis-server
----> Running in cdbe72bfdd6e
----> e91902a8ded0
Removing intermediate container cdbe72bfdd6e
Successfully built e91902a8ded0
```

镜像构建完成后，可以在本地镜像仓库中找到构建的 Redis 镜像。

```
$ sudo docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
------------	-----	----------	---------	------

```
ymdot/redis 0.0.1 e91902a8ded0 36 minutes ago 163 MB
```

## 9.2.4 测试 Redis 容器

与 Memcached 一样，Redis 定义的数据交换协议也十分简单，可以通过 Telnet 工具模拟对 Redis 的操作。

先创建一个基于 Redis 镜像的容器，并将 Redis 使用的 6379 端口绑定到外部网络，以便访问。

```
$ sudo docker run -d --name redis -p 6379:6379 ymdot/redis:0.0.1
aa13cb83e8c8d468a66256e10d7fa04fca0222fbec62cc2293bbfe5a35a7242d
```

通过 set、get、incrby、ttl 等 Redis 操作命令对 Redis 进行测试。这几个命令的使用方法如下。

```
set <key> <value>
get <key>
incrby <key> <value>
ttl <key>
```

其中，

❑ key: 缓存的名称。

❑ value: 缓存的数据。

通过 set 命令设置一条键值对。

```
set hello world
+OK
```

再通过 get 命令查看键值对的值。

```
get hello
$5
world
```

更新项目的值为数字。

```
set hello 56
+OK
get hello
$2
56
```



用 `incrby` 命令增加数字的值。

```
incrby hello 14
:70
get hello
$2
70
```

使用 `ttl` 命令查看特定项目的有效时间。

```
ttl hello
:-1
```

经过简单的测试，我们能够正常地连接运行在 Docker 容器中的 Redis，并能通过指令对 Redis 服务进行操作。所有操作的结果都符合我们的预期，这就表示我们构建的 Redis 镜像是可以使用的。

## 9.3 本章小结

在本章中，我们进行了在 Docker 中对 Memcached、Redis 两个程序的编译、安装、配置过程的实践，也掌握了使用 Dockerfile 编写和构建包含这两个程序的 Docker 镜像，并对它们进行了测试。

缓存工具是大型服务中提高运行效率、对抗大并发、减少用户等待时间等方面的重要支持者。掌握对缓存工具的搭建和使用方法，是开发者及运维人员的基本功之一。通过本章的实践，在今后的工作中，大家可以利用本章所学到的知识，以及我们自己编写的 Dockerfile，再利用 Docker 来提高搭建缓存工具的速度和稳定性。

# 第 10 章

## 动态处理程序

在 Web 领域，动态网页是非常重要的部分。动态网页是能够根据用户请求动态生成与动态更新网页的统称。和传统的静态网页相比，动态网页能够更好地展示网站提供的内容，特别是对于数据的更新，动态网页可以更快地将其展现给用户。

动态网页的生成，依赖于部署在 Web 服务器中的动态网页处理程序。Web 服务将收到的用户请求提交给动态处理程序，而动态处理程序根据用户的请求、存储在服务器中的数据及自身的业务逻辑，生成相应的网页，再由 Web 服务传递给用户。常见的用于处理 Web 请求的动态处理程序有 PHP、Java、Python、Node.js 等，本章就对这些动态处理程序在 Docker 中的安装和使用，以及如何构建这些动态处理程序的镜像进行讲解和实践。

### 10.1 Java

Java 作为一个跨平台的编程工具，被广泛应用在了各种设备上，在 Web 服务器领域也有一席之地。在 Docker 中，我们也能够使用 Java 程序运行和处理任务。

### 10.1.1 Java 简介

Java 是一种面向对象的编程语言，也是目前世界上使用范围最广的跨平台语言。它最初来自 Sun 公司为小型家用电器中的微型系统所开发的编程语言 Oak，在发展过程中，逐渐转向面向互联网应用的开发，并取名 Java。随着 Sun 公司被 Oracle（甲骨文）公司收购，Java 也并入 Oracle 公司旗下。图 10-1 展示了 Java 的标识。



图 10-1 Java 的标识

Java 借鉴了 C++ 的语法风格，摒弃了 C++ 中容易引起错误的指针类型，也抛弃了可以改写编译规则的运算符重载等语法。另外，Java 也改造了 C++ 的多重继承机制，改用接口和单继承来替换结构混乱的多重继承。

垃圾回收机制也是 Java 的精髓所在。其引以为豪的地方不仅是其采用了垃圾回收机制处理内存中不再被使用的部分，使开发者不需要过多地关注内存释放的问题。而且其比其他编程语言中的垃圾回收机制更加成熟。

Java 的名称来源于 Java 最初的开发者詹姆斯·高斯林和他的团队。他们喜欢在公司楼下的咖啡厅喝咖啡，而这个咖啡厅和其他大多数咖啡厅一样，常常用盛产咖啡的印尼爪哇岛做广告，于是就有了这个来自爪哇岛的名称，在 Java 的标识中我们也可以看到一杯香浓的咖啡。而在 Java 编译得到的 class 文件的前 32 个字节中，即 class 文件的 Magic 信息，也是由“Cafe Babe”的十六进制表示的：CA FE BA BE。

### 10.1.2 安装 Java

目前，Oracle 公司维护的 Java 分为两个版本，我们可以通过它们各自的开发工具进行区分：用于商业开发的是 JDK，开源版本是 OpenJDK。这里选择更加自由的 OpenJDK 作为实践对象。



Java 分为开发包和运行环境两部分，开发包（JDK）主要用于 Java 程序的开发，包含的工具及相关资源较多。我们通常只是使用 Docker 容器来运行 Java 程序，而非开发 Java 程序，所以这里只安装提交和占用都较小的 Java 运行环境（JRE）。

使用 Debian 系统镜像作为安装 Java 的基础镜像，基于系统镜像建立容器，并进入到容器之中。

```
$ sudo docker run -it --name java debian:jessie /bin/bash
root@483b5b6ffd42:/#
```

可以使用 Debian 的 APT 软件仓库安装 Java，这种安装过程比其他安装方式更快捷。

Java 在不断地更新迭代，这里选择目前较为常见的 OpenJDK 8 的 JRE 作为安装对象。在通过 APT 安装之前，需要执行 APT 的更新命令，获取软件的更新信息。

```
root@483b5b6ffd42:/# apt-get update
Get:1 http://security.debian.org jessie/updates InRelease [63.1 kB]
Ign http://httpredir.debian.org jessie InRelease
Get:2 http://httpredir.debian.org jessie-updates InRelease [142 kB]
Get:3 http://security.debian.org jessie/updates/main amd64 Packages [391 kB]
Get:4 http://httpredir.debian.org jessie Release.gpg [2373 B]
Get:5 http://httpredir.debian.org jessie Release [148 kB]
Get:6 http://httpredir.debian.org jessie-updates/main amd64 Packages [17.6 kB]
Get:7 http://httpredir.debian.org jessie/main amd64 Packages [9064 kB]
Fetched 9829 kB in 1min 39s (98.9 kB/s)
Reading package lists... Done
```

APT 仓库中还没有提供 OpenJDK 8 的 JRE 程序，通过 apt-cache search 命令只能搜索到 OpenJDK 7。

```
root@483b5b6ffd42:/# apt-cache search openjdk
jvm-7-avian-jre - lightweight virtual machine using the OpenJDK class library
icedtea-7-plugin - web browser plugin based on OpenJDK and IcedTea to execute Java applets
jtreg - Regression Test Harness for the OpenJDK platform
icedtea-7-jre-jamvm - Alternative JVM for OpenJDK, using JamVM
openjdk-7-dbg - Java runtime based on OpenJDK (debugging symbols)
openjdk-7-demo - Java runtime based on OpenJDK (demos and examples)
openjdk-7-doc - OpenJDK Development Kit (JDK) documentation
openjdk-7-jdk - OpenJDK Development Kit (JDK)
openjdk-7-jre - OpenJDK Java runtime, using Hotspot JIT
openjdk-7-jre-headless - OpenJDK Java runtime, using Hotspot JIT (headless)
openjdk-7-jre-lib - OpenJDK Java runtime (architecture independent libraries)
```

```
openjdk-7-jre-zero - Alternative JVM for OpenJDK, using Zero/Shark
openjdk-7-source - OpenJDK Development Kit (JDK) source files
openjdk-7-jre-dcevm - Alternative VM for OpenJDK 7 with enhanced class redefinition
uwsgi-plugin-jvm-openjdk-7 - Java plugin for uWSGI (OpenJDK 7)
uwsgi-plugin-jwsgi-openjdk-7 - JWSGI plugin for uWSGI (OpenJDK 7)
```

为了从 APT 中安装 Java, 将 OpenJDK 8 的安装源导入 APT 中, 再次执行软件源更新命令。

```
root@483b5b6ffd42:/# echo 'deb http://httpredir.debian.org/debian jessie-backports
main' > /etc/apt/sources.list.d/jessie-backports.list
root@483b5b6ffd42:/# apt-get update
Hit http://security.debian.org jessie/updates InRelease
Get:1 http://security.debian.org jessie/updates/main amd64 Packages [391 kB]
Ign http://httpredir.debian.org jessie InRelease
Hit http://httpredir.debian.org jessie-updates InRelease
Get:2 http://httpredir.debian.org jessie-backports InRelease [166 kB]
Hit http://httpredir.debian.org jessie Release.gpg
Hit http://httpredir.debian.org jessie Release
Get:3 http://httpredir.debian.org jessie-updates/main amd64 Packages [17.6 kB]
Get:4 http://httpredir.debian.org jessie-backports/main amd64 Packages [912 kB]
Get:5 http://httpredir.debian.org jessie/main amd64 Packages [9064 kB]
Fetched 10.6 MB in 47s (220 kB/s)
Reading package lists... Done
root@483b5b6ffd42:/# apt-cache search openjdk
jvm-7-avian-jre - lightweight virtual machine using the OpenJDK class library
icedtea-7-plugin - web browser plugin based on OpenJDK and IcedTea to execute Java
applets
jtreg - Regression Test Harness for the OpenJDK platform
icedtea-7-jre-jamvm - Alternative JVM for OpenJDK, using JamVM
openjdk-7-dbg - Java runtime based on OpenJDK (debugging symbols)
openjdk-7-demo - Java runtime based on OpenJDK (demos and examples)
openjdk-7-doc - OpenJDK Development Kit (JDK) documentation
openjdk-7-jdk - OpenJDK Development Kit (JDK)
openjdk-7-jre - OpenJDK Java runtime, using Hotspot JIT
openjdk-7-jre-headless - OpenJDK Java runtime, using Hotspot JIT (headless)
openjdk-7-jre-lib - OpenJDK Java runtime (architecture independent libraries)
openjdk-7-jre-zero - Alternative JVM for OpenJDK, using Zero/Shark
openjdk-7-source - OpenJDK Development Kit (JDK) source files
openjdk-7-jre-dcevm - Alternative VM for OpenJDK 7 with enhanced class redefinition
uwsgi-plugin-jvm-openjdk-7 - Java plugin for uWSGI (OpenJDK 7)
uwsgi-plugin-jwsgi-openjdk-7 - JWSGI plugin for uWSGI (OpenJDK 7)
openjdk-8-dbg - Java runtime based on OpenJDK (debugging symbols)
```



```

openjdk-8-demo - Java runtime based on OpenJDK (demos and examples)
openjdk-8-doc - OpenJDK Development Kit (JDK) documentation
openjdk-8-jdk - OpenJDK Development Kit (JDK)
openjdk-8-jdk-headless - OpenJDK Development Kit (JDK) (headless)
openjdk-8-jre - OpenJDK Java runtime, using Hotspot JIT
openjdk-8-jre-headless - OpenJDK Java runtime, using Hotspot JIT (headless)
openjdk-8-jre-jamvm - Alternative JVM for OpenJDK, using JamVM
openjdk-8-jre-zero - Alternative JVM for OpenJDK, using Zero/Shark
openjdk-8-source - OpenJDK Development Kit (JDK) source files
openjdk-8-jre-dcevm - Alternative VM for OpenJDK 8 with enhanced class redefinition

```

在新的关于 OpenJDK 的软件列表中，就可以看到 OpenJDK 8 了。

接着通过 apt-get 安装 Java。

```

root@483b5b6ffd42:/# apt-get install -y openjdk-8-jre
Reading package lists... Done
Building dependency tree... Done
The following extra packages will be installed:
  ca-certificates ca-certificates-java dbus dbus-x11 file fontconfig fontconfig-
config fonts-dejavu-core
  fonts-dejavu-extra gconf-service gconf2 gconf2-common gnome-mime-data hicolor-
icon-theme java-common krb5-locales
  libalgorithm-c3-perl libarchive-extract-perl libasound2 libasound2-data libasyncns0
libatk-wrapper-java
  libatk-wrapper-java-jni libatk1.0-0 libatk1.0-data libavahi-client3 libavahi-
common-data libavahi-common3
  libavahi-glib1 libbonobo2-0 libbonobo2-common libbsd0 libcairo2 libcanberra0
libcap-ng0 libcgi-fast-perl
  libcgi-pm-perl libclass-c3-perl libclass-c3-xs-perl libcpan-meta-perl libcups2
libdata-optlist-perl
  libdata-section-perl libdatatriel libdbus-1-3 libdbus-glib-1-2 libdrm-intel1 libdrm-
nouveau2 libdrm-radeon1 libdrm2
  libedit2 libelf1 libexpat1 libfcgi-perl libffi6 libflac8 libfontconfig1 libfreetype6
libgconf-2-4 libgconf2-4
  libgdbm3 libgdk-pixbuf2.0-0 libgdk-pixbuf2.0-common libgif4 libgl1-mesa-dri
libgl1-mesa-glx libglapi-mesa
  libglib2.0-0 libglib2.0-data libgmp10 libgnome-2-0 libgnome2-0 libgnome2-bin
libgnome2-common libgnomevfs2-0
  libgnomevfs2-common libgnomevfs2-extra libgnutls-deb0-28 libgraphite2-3 libgssapi-
krb5-2 libgtk2.0-0
  libgtk2.0-bin libgtk2.0-common libharfbuzz0b libhogweed2 libice6 libjasper1
libjbig0 libjpeg62-turbo libjson-c2

```



```

libk5crypto3 libkeyutils1 libkrb5-3 libkrb5support0 liblcms2-2 libldap-2.4-2
libllvm3.5 liblog-message-perl
liblog-message-simple-perl libltdl7 libmagic1 libmodule-build-perl libmodule-
pluggable-perl
libmodule-signature-perl libmro-compat-perl libnettle4 libnspr4 libnss3 libogg0
liborbit-2-0 libp11-kit0
libpackage-constants-perl libpango-1.0-0 libpangocairo-1.0-0 libpangoft2-1.0-0
libparams-util-perl libpciaccess0
libpcsc-lite1 libpixman-1-0 libpng12-0 libpod-latex-perl libpod-readme-perl
libpopt0 libpulse0 libpython-stdlib
libpython2.7-minimal libpython2.7-stdlib libregex-common-perl libsasl2-2
libsasl2-modules libsasl2-modules-db
libsm6 libsndfile1 libsoftware-license-perl libsqlite3-0 libssl1.0.0 libsub-
exporter-perl libsub-install-perl
libtasn1-6 libtdb1 libterm-ui-perl libtext-soundex-perl libtext-template-perl
libthai-data libthai0 libtiff5
libtxc-dxtn-s2tc0 libvorbis0a libvorbisenc2 libvorbisfile3 libwrap0 libx11-6
libx11-data libx11-xcb1 libxau6
libxcb-dri2-0 libxcb-dri3-0 libxcb-glx0 libxcb-present0 libxcb-render0 libxcb-shm0
libxcb-sync1 libxcb1
libxcomposite1 libxcursor1 libxdamage1 libxdmcp6 libxext6 libxfixes3 libxi6
libxinerama1 libxml2 libxrandr2
libxrender1 libxshmfence1 libxtst6 libxxf86vm1 mime-support openjdk-8-jre-headless
openssl perl perl-base
perl-modules psmisc python python-minimal python2.7 python2.7-minimal rename
sgml-base shared-mime-info tcpd ucf
x11-common xdg-user-dirs xml-core
Suggested packages:
gconf-defaults-service default-jre equivs libasound2-plugins alsa-utils libbonobo2-
bin libcanberra-gtk0
libcanberra-pulse cups-common desktop-base libgnomevfs2-bin gnutls-bin krb5-doc
krb5-user librsvg2-common gvfs
libjasper-runtime liblcms2-utils pciutils pcsd pulseaudio libsasl2-modules-otp
libsasl2-modules-ldap
libsasl2-modules-sql libsasl2-modules-gssapi-mit libsasl2-modules-gssapi-heimdal
icedtea-8-plugin
openjdk-8-jre-jamvm libnss-mdns fonts-ipafont-gothic fonts-ipafont-mincho ttf-wqy-
microhei ttf-wqy-zenhei
fonts-indic perl-doc libterm-readline-gnu-perl libterm-readline-perl-perl make
libb-lint-perl
libcanplus-dist-build-perl libcanplus-perl libfile-checktree-perl libobject-
accessor-perl python-doc python-tk
python2.7-doc binutils binfmt-support sgml-base-doc debhelper

```

Recommended packages:

libarchive-tar-perl

The following NEW packages will be installed:

ca-certificates ca-certificates-java dbus dbus-x11 file fontconfig fontconfig-  
config fonts-dejavu-core  
fonts-dejavu-extra gconf-service gconf2 gconf2-common gnome-mime-data hicolor-  
icon-theme java-common krb5-locales  
libalgorithm-c3-perl libarchive-extract-perl libasound2 libasound2-data  
libasyns0 libatk-wrapper-java  
libatk-wrapper-java-jni libatk1.0-0 libatk1.0-data libavahi-client3 libavahi-  
common-data libavahi-common3  
libavahi-glib1 libbonobo2-0 libbonobo2-common libbsd0 libcairo2 libcanberra0  
libcap-ng0 libcgi-fast-perl  
libcgi-pm-perl libclass-c3-perl libclass-c3-xs-perl libcpan-meta-perl libcups2  
libdata-optlist-perl  
libdata-section-perl libdatriel libdbus-1-3 libdbus-glib-1-2 libdrm-intel1 libdrm-  
nouveau2 libdrm-radeon1 libdrm2  
libedit2 libelf1 libexpat1 libfcgi-perl libffi6 libflac8 libfontconfig1  
libfreetype6 libgconf-2-4 libgconf2-4  
libgdbm3 libgdk-pixbuf2.0-0 libgdk-pixbuf2.0-common libgif4 libgl1-mesa-dri  
libgl1-mesa-glx libglapi-mesa  
libglib2.0-0 libglib2.0-data libgmp10 libgnome-2-0 libgnome2-0 libgnome2-bin  
libgnome2-common libgnomevfs2-0  
libgnomevfs2-common libgnomevfs2-extra libgnutls-deb0-28 libgraphite2-3 libgssapi-  
krb5-2 libgtk2.0-0  
libgtk2.0-bin libgtk2.0-common libharfbuzz0b libhogweed2 libice6 libjasper1  
libjbig0 libjpeg62-turbo libjson-c2  
libk5crypto3 libkeyutils1 libkrb5-3 libkrb5support0 liblcms2-2 libldap-2.4-2  
libllvm3.5 liblog-message-perl  
liblog-message-simple-perl libltdl7 libmagic1 libmodule-build-perl libmodule-  
pluggable-perl  
libmodule-signature-perl libmro-compat-perl libnettle4 libnspr4 libnss3 libogg0  
liborbit-2-0 libp11-kit0  
libpackage-constants-perl libpango-1.0-0 libpangocairo-1.0-0 libpangoft2-1.0-0  
libparams-util-perl libpciaccess0  
libpcsc-lite1 libpixman-1-0 libpng12-0 libpod-latex-perl libpod-readme-perl  
libpopt0 libpulse0 libpython-stdlib  
libpython2.7-minimal libpython2.7-stdlib libregex-common-perl libsasl2-2  
libsasl2-modules libsasl2-modules-db  
libsm6 libsndfile1 libsoftware-license-perl libsqlite3-0 libssl1.0.0 libsub-  
exporter-perl libsub-install-perl  
libtasn1-6 libtdb1 libterm-ui-perl libtext-soundex-perl libtext-template-perl



```

libthai-data libthai0 libtiff5
  libtxc-dxtn-s2tc0 libvorbis0a libvorbisenc2 libvorbisfile3 libwrap0 libx11-6
libx11-data libx11-xcb1 libxau6
  libxcb-dri2-0 libxcb-dri3-0 libxcb-glx0 libxcb-present0 libxcb-render0 libxcb-shm0
libxcb-sync1 libxcb1
  libxcompositel libxcursor1 libxdamage1 libxdmcp6 libxext6 libxfixed3 libxi6
libxineramal libxml2 libxrandr2
  libxrender1 libxshmfence1 libxtst6 libxxf86vm1 mime-support openjdk-8-jre
openjdk-8-jre-headless openssl perl
  perl-modules psmisc python python-minimal python2.7 python2.7-minimal rename
sgml-base shared-mime-info tcpd ucf
  x11-common xdg-user-dirs xml-core
The following packages will be upgraded:
  perl-base
1 upgraded, 196 newly installed, 0 to remove and 17 not upgraded.
Need to get 93.9 MB of archives.
After this operation, 346 MB of additional disk space will be used.
Get:1 http://security.debian.org/ jessie/updates/main libssl1.0.0 amd64 1.0.1t-1+
deb8u5 [1048 kB]
Get:2 http://httpredir.debian.org/debian/ jessie/main perl-base amd64 5.20.2-3+
deb8u6 [1229 kB]
Get:3 http://httpredir.debian.org/debian/ jessie/main libgdbm3 amd64 1.8.3-13.1 [30.0 kB]
Get:4 http://httpredir.debian.org/debian/ jessie/main libffi6 amd64 3.1-2+b2 [20.1 kB]
...
Processing triggers for sgml-base (1.26+nmu4) ...

```

安装完 Java 之后，可以通过 `java` 命令查看 Java 的版本。

```

root@483b5b6ffd42:/# java -version
openjdk version "1.8.0_102"
OpenJDK Runtime Environment (build 1.8.0_102-8u102-b14.1-1~bpo8+1-b14)
OpenJDK 64-Bit Server VM (build 25.102-b14, mixed mode)

```

我们看到，终端命令行中已经显示出了 OpenJDK 中 JRE 的版本，这就说明 OpenJDK Runtime Environment 已经安装成功了。

### 10.1.3 构建 Java 镜像

根据之前安装 Java 的过程，可以直接写出能够构建 Java 镜像的 Dockerfile。

```

# OpenJDK - JRE
# VERSION 0.0.1

```



```
# 基础镜像
FROM debian:jessie

# 维护者信息
MAINTAINER You Ming <youming@funcuter.org>

# 安装 Java
RUN echo 'deb http://httpredir.debian.org/debian jessie-backports main' > /etc/
apt/sources.list.d/jessie-backports.list \
    && apt-get update \
    && apt-get install -y openjdk-8-jre \
    && rm -rf /var/lib/apt/lists/*

# 通常来说, 使用 Java 镜像会指定不同的 Java 程序, 所以这里只打印 Java 的版本
CMD ["java", "-version"]
```

将编写好的 Dockerfile 保存到文件中, 再通过 `docker build` 命令构建 Java 镜像。

```
$ sudo docker build -t ymdot/openjdk:0.0.1 ./openjdk
Sending build context to Docker daemon 2.56 kB
Step 1 : FROM debian:jessie
---> 1b088884749b
Step 2 : MAINTAINER You Ming <youming@funcuter.org>
---> Using cache
---> 0b647e0ed3ee
Step 3 : RUN echo 'deb http://httpredir.debian.org/debian jessie-backports main' >
/etc/apt/sources.list.d/jessie-backports.list && apt-get update && apt-get
install -y openjdk-8-jre && rm -rf /var/lib/apt/lists/*
---> Running in 465e6fb3eda7
...
---> 160bb76a806d
Removing intermediate container 465e6fb3eda7
Step 4 : CMD java -version
---> Running in 4e9e9468532a
---> f39bdc25b727
Removing intermediate container 4e9e9468532a
Successfully built f39bdc25b727
```

构建镜像的过程完成后, 可以在本地镜像仓库中找到构建的 Java 镜像。

```
$ sudo docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
ymdot/openjdk	0.0.1	f39bdc25b727	5 minutes ago	498.1 MB
...				

## 10.1.4 测试 Java 容器

因为我们赋予前面构建的 Java 镜像的执行指令是 `java -version`，所以，如果直接通过前面构建的 Java 镜像运行容器，可以看到的就是容器中 OpenJDK 的版本信息。

```
$ sudo docker rrun -it --name java ymdot/openjdk:0.0.1
openjdk version "1.8.0_102"
OpenJDK Runtime Environment (build 1.8.0_102-8u102-b14.1-1~bpo8+1-b14)
OpenJDK 64-Bit Server VM (build 25.102-b14, mixed mode)
```

为了测试 Java 镜像，我们先编写一个简单的 Java 程序：每秒打印当前的系统时间戳到程序输出流。

```
package ymdot;

public class Test {

    public static void main(String args[]) {
        while (true) {
            // 打印时间戳
            System.out.println(System.currentTimeMillis() / 1000L);

            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                break;
            }
        }
    }
}
```

将代码保存成 `Test.java`，并通过 `javac` 命令将其编译成 `Test.class` 文件。由于 Java 具有跨平台性，所以不需要考虑编译 Java 程序的平台。也就是说，可以在 Windows、Mac、Linux 下编译 Java 程序，编译的程序都可以在 Docker 容器里的 Linux 环境下运行。

我们基于 Java 镜像新建一个新的容器，并把前面编译的 Java 程序通过数据卷的方式挂载到容器中。容器的启动命令直接采用运行这个 Java 程序的命令。

```
$ sudo docker run -t --name java -v /ymdot/Test.class:/ymdot/Test.class
ymdot/openjdk:0.0.1 java -cp / ymdot.Test
1476005206
1476005207
```

```
1476005208
1476005209
1476005210
...
```

通过这种方式启动了 Java 容器后，可以看到随着时间的递增，时间戳被逐一打印出来，这就说明可以通过前面构建的 Java 镜像及它创建的容器运行 Java 程序。

## 10.2 PHP

PHP 是一门悠久的 Web 开发语言，诞生已超过 20 年，经久不衰。在 Docker 中，我们也能够使用独立的容器运行 PHP。

### 10.2.1 PHP 简介

PHP (Hypertext Preprocessor) 是一种开源的脚本语言。它从诞生之初，就确定了其主要用于动态网页处理的特性。

PHP 的语言特性非常松散，开发者可以以很少的代码实现丰富的功能。在运算细节上，也不需要过多考虑变量类型等问题。这使利用 PHP 开发程序的过程非常有效率，尤其适用于经常需要进行改动的 Web 页面。另外，PHP 作为脚本语言，编写完成后不需要进行编译就能够直接通过解析器运行，这也能够提升开发速度。

PHP 就是为了 Web 开发设计的，所以在二十多年的发展中，积累了大量能够帮助和辅助 Web 开发者的功能、技巧。这也使 PHP 在 Web 服务的动态解析语言界，有着响亮的名号。目前，PHP 已经是全世界最受欢迎的 Web 动态处理语言之一，有近 3 亿的网站使用 PHP 进行 Web 页面的处理，包括我们熟知的 Facebook、淘宝、微博等许多大型网站在内。这些网站不但体量巨大，而且面向上亿的访问者。这足以说明 PHP 对大型网站的请求的处理能力。图 10-2 展示了 PHP 的标识。



图 10-2 PHP 的标识



PHP 在不断地演进中吸收了不同语言的优点，形成了自己的语法特色。目前，PHP 已经能够同时支持面向过程和面向对象的程序逻辑。在一些场景下，函数式编程也能在 PHP 中实现。PHP 通过强大而丰富的扩展库，为开发者提供了非常丰富的其他功能，这就使开发者可以更专注于自己的业务逻辑和 Web 页面功能的实现。

目前，PHP 已经迭代到了第 7 代，采用了性能更高的 Zend Engine 3 引擎，并加入了底层优化和语法抽象树等特性。PHP 的目标是在高效开发的语法基础上，更接近编译型语言的运算效率。

## 10.2.2 安装 PHP

目前最新的 PHP 稳定版本为 7.0.11，我们使用此版本进行 PHP 安装的实践。对于运行 PHP 的系统环境，我们选择 Debian 系统。

首先，在 Docker 中建立一个基于 Debian 系统镜像的容器，并进入到容器之中。

```
$ sudo docker run --name php -it debian:jessie /bin/bash
root@8a466e272abd:/#
```

在安装 PHP 之前，先通过 Debian 系统携带的 APT 软件仓库下载和安装 PHP 所需的几个依赖库。因为 PHP 充分使用了其他软件或库来完成自己的功能，所以 PHP 能够支持的功能非常丰富，这就造成了在安装 PHP 前需要安装的依赖库很多。不但需要安装编译 PHP 所使用的 autoconf、file、g++、gcc、libc-dev、make、pkg-config、re2c，还需要安装 ca-certificates、curl、libedit2、libsqlite3-0、libxml2、xz-utils。其中，re2c 主要用于 PHP 语法解析程序的生成。

在安装这些依赖库之前，先对 APT 软件仓库中的软件源进行更新。

```
root@8a466e272abd:/# apt-get update
Ign http://httpredir.debian.org jessie InRelease
Get:1 http://httpredir.debian.org jessie-updates InRelease [142 kB]
Get:2 http://security.debian.org jessie/updates InRelease [63.1 kB]
Get:3 http://httpredir.debian.org jessie Release.gpg [2373 B]
Get:4 http://httpredir.debian.org jessie Release [148 kB]
Get:5 http://httpredir.debian.org jessie-updates/main amd64 Packages [17.6 kB]
Get:6 http://security.debian.org jessie/updates/main amd64 Packages [391 kB]
Get:7 http://httpredir.debian.org jessie/main amd64 Packages [9064 kB]
Fetched 9829 kB in 13s (739 kB/s)
Reading package lists... Done
```

更新完软件源之后,就可以通过 `apt-get` 命令进行安装。部分类库还要安装它们的开发文件,以便 PHP 编译时进行连接,如 `libcurl4-openssl-dev`、`libedit-dev`、`libsqlite3-dev`、`libssl-dev`、`libxml2-dev`。

```
root@8a466e272abd:/# apt-get install -y autoconf file g++ gcc libc-dev make pkg-config
re2c ca-certificates curl libedit2 libsqlite3-0 libxml2 xz-utils libcurl4-openssl-dev
libedit-dev libsqlite3-dev libssl-dev libxml2-dev
Reading package lists... Done
Building dependency tree... Done
Note, selecting 'libc6-dev' instead of 'libc-dev'
The following extra packages will be installed:
  automake autotools-dev binutils cpp cpp-4.9 g++-4.9 gcc-4.9 krb5-locales
  libalgorithm-c3-perl
  libarchive-extract-perl libasan1 libatomic1 libbsd-dev libbsd0 libc-dev-bin libc6
  libcgi-fast-perl libcgi-pm-perl
  libcilkrts5 libclass-c3-perl libclass-c3-xs-perl libcloog-isl4 libcpan-meta-perl
  libcurl3 libdata-optlist-perl
  libdata-section-perl libfcgi-perl libffi6 libgcc-4.9-dev libgdbm3 libglib2.0-0
  libglib2.0-data libgmp10
  libgnutls-deb0-28 libgomp1 libgssapi-krb5-2 libhogweed2 libidn11 libisl10 libitm1
  libk5crypto3 libkeyutils1
  libkrb5-3 libkrb5support0 libldap-2.4-2 liblog-message-perl liblog-message-
  simple-perl liblsan0 libmagic1
  libmodule-build-perl libmodule-pluggable-perl libmodule-signature-perl libmpc3
  libmpfr4 libmro-compat-perl
  libnettle4 libp11-kit0 libpackage-constants-perl libparams-util-perl libpod-latex-
  perl libpod-readme-perl
  libquadmath0 libregex-common-perl librtmp1 libsass2-2 libsass2-modules libsass2-
  modules-db libsigsegv2
  libsoftware-license-perl libssh2-1 libssl-doc libssl1.0.0 libstdc++-4.9-dev
  libsub-exporter-perl
  libsub-install-perl libtasn1-6 libterm-ui-perl libtext-soundex-perl libtext-
  template-perl libtinfo-dev libtsan0
  libubsan0 linux-libc-dev m4 manpages manpages-dev openssl perl perl-base
  perl-modules rename sgml-base
  shared-mime-info xdg-user-dirs xml-core zlib1g-dev
Suggested packages:
  autoconf-archive gnu-standards autoconf-doc libtool gettext binutils-doc cpp-doc
  gcc-4.9-locales g++-multilib
  g++-4.9-multilib gcc-4.9-doc libstdc++6-4.9-dbg gcc-multilib flex bison gdb gcc-doc
  gcc-4.9-multilib libgcc1-dbg
  libgomp1-dbg libitm1-dbg libatomic1-dbg libasan1-dbg liblsan0-dbg libtsan0-dbg
```

```
libubsan0-dbg libcilkrts5-dbg
libquadmath0-dbg glibc-doc libcurl4-doc libcurl3-dbg libidn11-dev libkrb5-dev
libldap2-dev librtmp-dev
libssh2-1-dev gnutls-bin krb5-doc krb5-user libsasl2-modules-otp libsasl2-modules-
ldap libsasl2-modules-sql
libsasl2-modules-gssapi-mit libsasl2-modules-gssapi-heimdal sqlite3-doc libstdc+
+-4.9-doc make-doc man-browser
perl-doc libterm-readline-gnu-perl libterm-readline-perl-perl libb-lint-perl
libcpanplus-dist-build-perl
libcpanplus-perl libfile-checktree-perl libobject-accessor-perl sgml-base-doc
debhelper
Recommended packages:
libarchive-tar-perl
The following NEW packages will be installed:
autoconf automake autotools-dev binutils ca-certificates cpp cpp-4.9 curl file g++
g++-4.9 gcc gcc-4.9
krb5-locales libalgorithm-c3-perl libarchive-extract-perl libasan1 libatomic1
libbsd-dev libbsd0 libc-dev-bin
libc6-dev libcgi-fast-perl libcgi-pm-perl libcilkrts5 libclass-c3-perl libclass-
c3-xs-perl libcloog-isl4
libcpan-meta-perl libcurl3 libcurl4-openssl-dev libdata-optlist-perl libdata-
section-perl libedit-dev libedit2
libfcgi-perl libffi6 libgcc-4.9-dev libgdbm3 libglib2.0-0 libglib2.0-data libgmp10
libgnutls-deb0-28 libgomp1
libgssapi-krb5-2 libhogweed2 libidn11 libisl10 libitm1 libk5crypto3 libkeyutils1
libkrb5-3 libkrb5support0
libldap-2.4-2 liblog-message-perl liblog-message-simple-perl liblsan0 libmagic1
libmodule-build-perl
libmodule-pluggable-perl libmodule-signature-perl libmpc3 libmpfr4 libmpc-compat-
perl libnettle4 libp11-kit0
libpackage-constants-perl libparams-util-perl libpod-latex-perl libpod-readme-
perl libquadmath0
libregex-common-perl librtmp1 libsasl2-2 libsasl2-modules libsasl2-modules-db
libsigsegv2
libsoftware-license-perl libsqlite3-0 libsqlite3-dev libssh2-1 libssl-dev libssl-
doc libssl1.0.0
libstdc++-4.9-dev libsub-exporter-perl libsub-install-perl libtasn1-6 libterm-ui-
perl libtext-soundex-perl
libtext-template-perl libtinfo-dev libtsan0 libubsan0 libxml2 libxml2-dev linux-
libc-dev m4 make manpages
manpages-dev openssl perl perl-modules pkg-config re2c rename sgml-base shared-
mime-info xdg-user-dirs xml-core
```



```

xz-utils zlib1g-dev
The following packages will be upgraded:
  libc6 perl-base
2 upgraded, 113 newly installed, 0 to remove and 16 not upgraded.
Need to get 44.6 MB/80.9 MB of archives.
After this operation, 238 MB of additional disk space will be used.
Get:1 http://security.debian.org/ jessie/updates/main libssl-dev amd64 1.0.1t-1+deb8u5 [1283 kB]
Get:2 http://httpredir.debian.org/debian/ jessie/main libgcc-4.9-dev amd64 4.9.2-10 [2066 kB]
...
Processing triggers for sgml-base (1.26+nmu4) ...

```

接着，从 PHP 官网的代码镜像中下载 PHP 源代码。

```

root@8a466e272abd:/# curl -fSL
"https://secure.php.net/get/php-7.0.11.tar.gz/from/this/mirror" -o php.tar.gz
%   Total    % Received % Xferd Average Speed   Time    Time     Time    Current
                                 Dload  Upload Total   Spent    Left     Speed
100   1        0    1    0    0      0    0  --:--:--   0:00:01  --:--:--    0
100  184    100   184    0    0     58    0   0:00:03   0:00:03  --:--:--   249
100 18.1M   100 18.1M    0    0  95359    0   0:03:19   0:03:19  --:--:--  168k

```

再对 PHP 的源代码包进行解压操作。

```

root@8a466e272abd:/# tar xzf php.tar.gz

```

解压完成后进入到代码目录中，之后就可以进行 PHP 的编译配置、编译和安装了。

```

root@8a466e272abd:/php-7.0.11# ./configure --disable-cgi --enable-ftp --enable-
mbstring --enable-mysqld --with-curl --with-libedit --with-openssl --with-zlib
checking for grep that handles long lines and -e... /bin/grep
checking for egrep... /bin/grep -E
checking for a sed that does not truncate output... /bin/sed
checking build system type... x86_64-unknown-linux-gnu
checking host system type... x86_64-unknown-linux-gnu
checking target system type... x86_64-unknown-linux-gnu
checking for cc... cc
checking whether the C compiler works... yes
checking for C compiler default output file name... a.out
checking for suffix of executables...
checking whether we are cross compiling... no
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether cc accepts -g... yes

```

```
checking for cc option to accept ISO C89... none needed
checking how to run the C preprocessor... cc -E
checking for icc... no
checking for suncc... no
checking whether cc understands -c and -o together... yes
checking how to run the C preprocessor... cc -E
checking for ANSI C header files... yes
checking for sys/types.h... yes
checking for sys/stat.h... yes
checking for stdlib.h... yes
checking for string.h... yes
checking for memory.h... yes
checking for strings.h... yes
checking for inttypes.h... yes
checking for stdint.h... yes
checking for unistd.h... yes
checking minix/config.h usability... no
checking minix/config.h presence... no
checking for minix/config.h... no
checking whether it is safe to define __EXTENSIONS__... yes
checking whether ln -s works... yes
checking for system library directory... lib
checking whether to enable runpaths... yes
checking if compiler supports -R... no
checking if compiler supports -Wl,-rpath,... yes
checking for gawk... no
checking for nawk... nawk
checking if nawk is broken... no
checking for bison... no
checking for byacc... no
checking for bison version... invalid
configure: WARNING: This bison version is not supported for regeneration of the Zend/PHP
parsers (found: none, min: 204, excluded: ).
checking for re2c... re2c
checking for re2c version... 0.13.5 (ok)
checking whether to enable computed goto gcc extension with re2c... no
checking whether to force non-PIC code in shared modules... no
checking whether /dev/urandom exists... yes
checking whether /dev/arandom exists... no
checking for global register variables support... yes
checking for pthreads_cflags... -pthread
checking for pthreads_lib...
```



```

...
root@8a466e272abd:/php-7.0.11# make && make install /
bin/bash /php-7.0.11/libtool --silent --preserve-dup-deps --mode=compile cc
-DZEND_ENABLE_STATIC_TSRMLS_CACHE=1 -Iext/standard/ -I/php-7.0.11/ext/standard/
-DPHP_ATOM_INC -I/php-7.0.11/include -I/php-7.0.11/main -I/php-7.0.11 -I/php-
7.0.11/ext/date/lib -I/usr/include/libxml2 -I/php-7.0.11/ext/mbstring/oniguruma
-I/php-7.0.11/ext/mbstring/libmbfl -I/php-7.0.11/ext/mbstring/libmbfl/mbfl -I/php-
7.0.11/ext/sqlite3/libsqlite -I/php-7.0.11/TSRM -I/php-7.0.11/Zend -I/usr/
include -g -O2 -fvisibility=hidden -c /php-7.0.11/ext/standard/info.c -o ext/
standard/info.lo
...
Installing shared extensions:      /usr/local/lib/php/extensions/no-debug-non-zts-
20151012/
Installing PHP CLI binary:         /usr/local/bin/
Installing PHP CLI man page:       /usr/local/php/man/man1/
Installing phpdbg binary:          /usr/local/bin/
Installing phpdbg man page:        /usr/local/php/man/man1/
Installing build environment:      /usr/local/lib/php/build/
Installing header files:           /usr/local/include/php/
Installing helper programs:        /usr/local/bin/
    program: phize
    program: php-config
Installing man pages:              /usr/local/php/man/man1/
    page: phize.1
    page: php-config.1
Installing PEAR environment:       /usr/local/lib/php/
[PEAR] Archive_Tar    - installed: 1.4.0
[PEAR] Console_Getopt - installed: 1.4.1
[PEAR] Structures_Graph - installed: 1.1.1
[PEAR] XML_Util       - installed: 1.3.0
[PEAR] PEAR           - installed: 1.10.1
Wrote PEAR system config file at: /usr/local/etc/pear.conf
You may want to add: /usr/local/lib/php to your php.ini include_path
/php-7.0.11/build/shtool install -c ext/phar/phar.phar /usr/local/bin
ln -s -f phar.phar /usr/local/bin/phar
Installing PDO headers:            /usr/local/include/php/ext/pdo/

```

至此，PHP 就安装完成了，我们可以通过 `php -v` 命令查看 PHP 的版本和相关信息。

```

root@8a466e272abd:/php-7.0.11# php -v
PHP 7.0.11 (cli) (built: Oct 11 2016 05:14:24) ( NTS )
Copyright (c) 1997-2016 The PHP Group
Zend Engine v3.0.0, Copyright (c) 1998-2016 Zend Technologies

```



## 10.2.3 构建 PHP 镜像

根据安装 PHP 的实践，把构建 PHP 镜像的过程写成对应的 Dockerfile。

```
# PHP
# VERSION 0.0.1

# 基础镜像
FROM debian:jessie

# 维护者信息
MAINTAINER You Ming <youming@funcuter.org>

# 安装 PHP
RUN set -xe \
    && buildDeps=" \
        autoconf \
        file \
        g++ \
        gcc \
        libc-dev \
        make \
        pkg-config \
        re2c \
        ca-certificates \
        curl \
        libedit2 \
        libsqlite3-0 \
        libxml2 \
        xz-utils \
        libcurl4-openssl-dev \
        libedit-dev \
        libsqlite3-dev \
        libssl-dev \
        libxml2-dev \
    " \
    && apt-get update && apt-get install -y $buildDeps --no-install-recommends \
    && rm -rf /var/lib/apt/lists/* \
# 下载 PHP 源码
    && curl -L "https://secure.php.net/get/php-7.0.11.tar.gz/from/this/mirror" -o \
php.tar.gz \
    && mkdir -p /usr/src/php \
    && tar -xzf php.tar.gz -C /usr/src/php --strip-components=1 \
```

```

&& rm php.tar.gz \
&& cd /usr/src/php \
# 配置和安装
&& ./configure \
    --disable-cgi \
    --enable-ftp \
    --enable-mbstring \
    --enable-mysqld \
    --with-curl \
    --with-libedit \
    --with-openssl \
    --with-zlib \
&& make \
&& make install

```

# 打印 PHP 的版本信息

```
CMD ["php", "-v"]
```

将 Dockerfile 保存到文件中，之后通过 docker build 命令构建 PHP 镜像。

```

$ sudo docker build -t ymdot/php:0.0.1 ./php
Sending build context to Docker daemon 3.072 kB
Step 1 : FROM debian:jessie
----> 1b088884749b
Step 2 : MAINTAINER You Ming <youming@funcuter.org>
----> Using cache
----> 0b647e0ed3ee
Step 3 : RUN set -xe && buildDeps=" autoconf      file      g++      gcc
libc-dev      make      pkg-config re2c      ca-certificates curl
libedit2      libsqlite3-0 libxml2    xz-utils libcurl4-openssl-dev
      libedit-dev      libsqlite3-dev      libssl-dev      libxml2-dev      "
&& apt-get update && apt-get install -y $buildDeps --no-install-recommends
----> Running in 321ce254b95a
...
----> 59e1d83f2f57
Removing intermediate container 1aec08783d0a
Step 4 : CMD php -v
----> Running in fda4d727ab5b
----> e5245bef613f
Removing intermediate container fda4d727ab5b
Successfully built e5245bef613f

```

构建镜像之后，可以从本地镜像仓库中找到构建好的 PHP 镜像。

```
$ sudo docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
ymdot/php	0.0.1	e5245bef613f	22 hours ago	730.2 MB
...				

## 10.2.4 测试 PHP 容器

我们可以直接通过前面构建的 PHP 镜像创建并运行容器。

```
$ sudo docker run -t --name php ymdot/php:0.0.1
PHP 7.0.11 (cli) (built: Oct 12 2016 06:24:24) ( NTS )
Copyright (c) 1997-2016 The PHP Group
Zend Engine v3.0.0, Copyright (c) 1998-2016 Zend Technologies
```

因为在构建的 PHP 镜像里设置默认的启动命令为 `php -v`，所以运行相应的容器时终端上会打印出 PHP 的版本信息。

为了测试 PHP 程序在容器中的运行情况，我们先编写一个简单的 PHP 程序。这里同样以打印时间为例，编写一个能够逐渐打印时间到程序输出的 PHP 程序，参考代码样例如下。

```
<?php

while (true) {
    // 显示当期时间
    echo date('Y-m-d H:i:s');
    echo "\n";

    sleep(1);
}
```

将编写的 PHP 代码保存为 `time.php`，再将其放置到可以被挂载到 Docker 中的目录下，就可以运行此 PHP 程序的命令，并基于 PHP 镜像创建和运行容器。在创建容器的时候，不要忘记挂载 `time.php` 文件到容器之中。

```
$ sudo docker run -t --name php -v /php/time.php:/time.php ymdot/php:0.0.1 php
/time.php
2016-10-12 18:05:31
2016-10-12 18:05:32
2016-10-12 18:05:33
2016-10-12 18:05:34
2016-10-12 18:05:35
```



2016-10-12 18:05:36

容器启动之后，我们可以看到，当前的时间慢慢显示到了终端中，这就说明了 PHP 程序已经在容器中正常运行，即我们刚刚通过 Dockerfile 构建的 PHP 镜像是能够正常使用的。

## 10.3 Python

Python 是一门卓有建树的脚本语言，拥有丰富的功能，也具有脚本语言快速编写的特性，已经被广泛应用于众多领域。在 Docker 中，我们也能在容器中运行 Python 程序。

### 10.3.1 Python 简介

Python 与其他脚本语言类似，语法并不复杂，并且支持动态类型、垃圾回收等功能。但 Python 比较有特色的地方在于，其采用了强制缩进的方式来定义语法块。

通常，脚本语言用于补足编译型语言开发和运行较烦琐的问题，所以，脚本语言通常弱化了数据类型，同时加入了更多的功能性函数、方法或语法糖。但是脚本语言由于其特性，不会用作大型应用开发，所以常规的脚本语言在一些大型项目所需的功能支持上是缺失的。但是，Python 在多任务管理、线程控制、网络编程方面都提供了强有力的支持，这就决定了它可以适用于大型的作业系统中。图 10-3 展示了 Python 的标识。



图 10-3 Python 的标识

而且，Python 在语法上支持面向过程的编程、面向对象的编程、面向切面的编程及函数式编程。这几乎囊括了目前所有常用的编程方式，让习惯了不同编程方式的开发者，可以共同使用 Python 进行合作。

Python 官方提供了用 C 语言编写的解析运行程序，通过这个程序 Python 脚本就能运行于计算机中的程序。除了由 C 语言实现的 Python 解析程序，在 Python 的发展过程中，还逐渐出现了许多遵循 Python 语法，却以其他的语言实现的 Python 解析程序，包

括由 Java 实现的 Jython、由 .Net 实现的 IronPython，甚至还有本身就以 Python 实现的 PyPy。这些解析程序的出现，让 Python 适用和运行的范围变得更加广阔。

## 10.3.2 安装 Python

要在 Docker 中运行 Python，需要先创建一个新的容器来安装 Python 解析程序。这里使用 Debian 系统镜像来创建新的容器，将其命名为 python 并连接和进入到容器之中。

```
$ sudo docker run -it --name python debian:jessie /bin/bash
root@d9507a3c654b:/#
```

我们选择以源码方式来安装 Python 解析程序，这样就能运行最新版本的 Python。由于要进行源码编译，所以在安装 Python 解析程序之前，需要保证容器中含有能够帮助我们编译的工具及编译 Python 所必需的依赖库。

我们通过 Debian 镜像中含有的 APT 软件仓库来安装编译 Python 所必需的软件和依赖库。在使用 APT 安装这些软件之前，需要先更新 APT 软件仓库中的安装源信息。

```
root@d9507a3c654b:/# apt-get update
Ign http://httpredir.debian.org jessie InRelease
Get:1 http://httpredir.debian.org jessie-updates InRelease [142 kB]
Get:2 http://security.debian.org jessie/updates InRelease [63.1 kB]
Get:3 http://httpredir.debian.org jessie Release.gpg [2373 B]
Get:4 http://httpredir.debian.org jessie Release [148 kB]
Get:5 http://httpredir.debian.org jessie/main amd64 Packages [9064 kB]
Get:6 http://httpredir.debian.org jessie-updates/main amd64 Packages [17.6 kB]
Get:7 http://security.debian.org jessie/updates/main amd64 Packages [392 kB]
Fetched 9829 kB in 16s (588 kB/s)
Reading package lists... Done
```

接着，就可以安装编译 Python 需要的软件和依赖库了。

```
root@d9507a3c654b:/# apt-get install -y wget gcc make autoconf tcl-dev tk-dev
Reading package lists... Done
Building dependency tree... Done
The following extra packages will be installed:
  automake autotools-dev binutils build-essential bzip2 ca-certificates cpp cpp-4.9
  dpkg-dev fakeroot
  fontconfig-config fonts-dejavu-core g++ g++-4.9 gcc-4.9 libalgorithm-c3-perl
  libalgorithm-diff-perl
  libalgorithm-diff-xs-perl libalgorithm-merge-perl libarchive-extract-perl libasan1
  libatomic1
```

```

libbsd0 libc-dev-bin libc6 libc6-dev libcgi-fast-perl libcgi-pm-perl libcilkrts5
libclass-c3-perl
libclass-c3-xs-perl libcloog-isl4 libcpan-meta-perl libdata-optlist-perl libdata-
section-perl
libdpkg-perl libdrm-intel1 libdrm-nouveau2 libdrm-radeon1 libdrm2 libedit2 libelf1
libexpat1
libexpat1-dev libfakeroot libfcgi-perl libffi6 libfile-fcntllock-perl libfontconfig1
libfontconfig1-dev libfontenc1 libfreetype6 libfreetype6-dev libgcc-4.9-dev libgdbm3
libgl1-mesa-dri libgl1-mesa-glx libglapi-mesa libgl2.0-0 libgl2.0-data libgmp10
libgnutls-deb0-28 libgomp1 libhogweed2 libice-dev libice6 libicu52 libidn11
libisl10 libitm1
libllvm3.5 liblog-message-perl liblog-message-simple-perl liblsan0 libmodule-
build-perl
libmodule-pluggable-perl libmodule-signature-perl libmpc3 libmpfr4 libmro-compat-
perl libnettle4
libp11-kit0 libpackage-constants-perl libparams-util-perl libpciaccess0 libpng12-0
libpng12-dev
libpod-latex-perl libpod-readme-perl libpsl0 libpthread-stubs0-dev libquadmath0
libregex-common-perl libsigsegv2 libsm-dev libsm6 libsoftware-license-perl
libssl1.0.0
libstdc++-4.9-dev libsub-exporter-perl libsub-install-perl libtasn1-6 libtcl8.6
libterm-ui-perl
libtext-soundex-perl libtext-template-perl libtimedate-perl libtk8.6 libtsan0
libtxc-dxtn-s2tc0
libubsan0 libutempter0 libx11-6 libx11-data libx11-dev libx11-doc libx11-xcb1
libxau-dev libxau6
libxaw7 libxcb-dri2-0 libxcb-dri3-0 libxcb-glx0 libxcb-present0 libxcb-shape0
libxcb-sync1 libxcb1
libxcb1-dev libxcomposit1 libxdamage1 libxdmcp-dev libxdmcp6 libxext-dev libxext6
libxf86fixes3
libxft-dev libxft2 libxi6 libxinerama1 libxml2 libxmu6 libxmuu1 libxpm4 libxrandr2
libxrender-dev
libxrender1 libxshmfence1 libxss-dev libxss1 libxt-dev libxt6 libxtst6 libxv1
libxxf86dga1
libxxf86vm1 linux-libc-dev m4 manpages manpages-dev openssl patch perl perl-base
perl-modules
pkg-config rename sgml-base shared-mime-info tcl tcl8.6 tcl8.6-dev tk tk8.6 tk8.6-dev
ucf
x11-common x11-utils x11proto-core-dev x11proto-input-dev x11proto-kb-dev x11proto-
render-dev
x11proto-scrnsaver-dev x11proto-xext-dev xbitmaps xdg-user-dirs xml-core xorg-
sgml-doctools xterm

```



```
xtrans-dev xz-utils zlib1g-dev
Suggested packages:
  autoconf-archive gnu-standards autoconf-doc libtool gettext binutils-doc bzip2-doc
  cpp-doc
  gcc-4.9-locales debian-keyring g++-multilib g++-4.9-multilib gcc-4.9-doc libstdc++
  6-4.9-dbg
  gcc-multilib flex bison gdb gcc-doc gcc-4.9-multilib libgcc1-dbg libgomp1-dbg
  libitm1-dbg
  libatomic1-dbg libasan1-dbg liblsan0-dbg libtsan0-dbg libubsan0-dbg libcilkrts5-dbg
  libquadmath0-dbg glibc-doc gnutls-bin libice-doc pciutils libsm-doc libstdc++-
  4.9-doc libxcb-doc
  libxext-doc libxt-doc make-doc man-browser ed diffutils-doc perl-doc libterm-
  readline-gnu-perl
  libterm-readline-perl-perl libb-lint-perl libcpanplus-dist-build-perl libcpanplus-
  perl
  libfile-checktree-perl libobject-accessor-perl sgml-base-doc tcl-doc tcl-
  tclreadline tcl8.6-doc
  tk-doc tk8.6-doc mesa-utils debhelper xfonts-cyrillic
Recommended packages:
  libarchive-tar-perl
The following NEW packages will be installed:
  autoconf automake autotools-dev binutils build-essential bzip2 ca-certificates cpp
  cpp-4.9 dpkg-dev
  fakeroot fontconfig-config fonts-dejavu-core g++ g++-4.9 gcc gcc-4.9 libalgorithm-
  c3-perl
  libalgorithm-diff-perl libalgorithm-diff-xs-perl libalgorithm-merge-perl libarchive-
  extract-perl
  libasan1 libatomic1 libbsd0 libc-dev-bin libc6-dev libcgi-fast-perl libcgi-pm-perl
  libcilkrts5
  libclass-c3-perl libclass-c3-xs-perl libcloog-isl4 libcpan-meta-perl libdata-
  optlist-perl
  libdata-section-perl libdpkg-perl libdrm-intel1 libdrm-nouveau2 libdrm-radeon1
  libdrm2 libedit2
  libelf1 libexpat1 libexpat1-dev libfakeroot libfcgi-perl libffi6 libfile-
  fcntllock-perl
  libfontconfig1 libfontconfig1-dev libfontenc1 libfreetype6 libfreetype6-dev
  libgcc-4.9-dev libgdbm3
  libgl1-mesa-dri libgl1-mesa-glx libglapi-mesa libglib2.0-0 libglib2.0-data
  libgmp10
  libgnutls-deb0-28 libgomp1 libhogweed2 libice-dev libice6 libicu52 libidn11
  libisl10 libitm1
  libllvm3.5 liblog-message-perl liblog-message-simple-perl liblsan0 libmodule-
```

```

build-perl
  libmodule-pluggable-perl libmodule-signature-perl libmpc3 libmpfr4 libmro-compat-
perl libnettle4
  libp11-kit0 libpackage-constants-perl libparams-util-perl libpciaccess0 libpng12-0
libpng12-dev
  libpod-latex-perl libpod-readme-perl libpsl0 libpthread-stubs0-dev libquadmath0
  libregex-common-perl libsigsegv2 libsm-dev libsm6 libsoftware-license-perl
libssl1.0.0
  libstdc++-4.9-dev libsub-exporter-perl libsub-install-perl libtasn1-6 libtcl8.6
libterm-ui-perl
  libtext-soundex-perl libtext-template-perl libtimedate-perl libtk8.6 libtsan0
libtxc-dxtn-s2tc0
  libubsan0 libutempter0 libx11-6 libx11-data libx11-dev libx11-doc libx11-xcb1
libxau-dev libxau6
  libxaw7 libxcb-dri2-0 libxcb-dri3-0 libxcb-glx0 libxcb-present0 libxcb-shape0
libxcb-sync1 libxcb1
  libxcb1-dev libxcompositel libxdamage1 libxdmcp-dev libxdmcp6 libxext-dev libxext6
libxfixes3
  libxft-dev libxft2 libxi6 libxineramal libxml2 libxmu6 libxmuu1 libxpm4 libxrandr2
libxrender-dev
  libxrender1 libxshmfence1 libxss-dev libxss1 libxt-dev libxt6 libxtst6 libxv1
libxxf86dgal
  libxxf86vm1 linux-libc-dev m4 make manpages manpages-dev openssl patch perl
perl-modules pkg-config
  rename sgml-base shared-mime-info tcl tcl-dev tcl8.6 tcl8.6-dev tk tk-dev tk8.6
tk8.6-dev ucf wget
  x11-common x11-utils x11proto-core-dev x11proto-input-dev x11proto-kb-dev x11proto-
render-dev
  x11proto-scrnsaver-dev x11proto-xext-dev xbitmaps xdg-user-dirs xml-core xorg-
sgml-doctools xterm
  xtrans-dev xz-utils zlib1g-dev
The following packages will be upgraded:
  libc6 perl-base
2 upgraded, 195 newly installed, 0 to remove and 16 not upgraded.
Need to get 110 MB of archives.
After this operation, 351 MB of additional disk space will be used.
...

```

目前, Python 已经迭代到了第 3 代, 最新的 Python 稳定版本为 3.5.2, 我们以此版本的 Python 的解析程序作为样例进行 Python 解析程序的安装。

在编译 Python 解析程序之前, 先从互联网上下载 Python 解析程序的源代码, 并把源代码包进行解压。



```

root@d9507a3c654b:/# wget -O python.tgz "http://www.python.org/ftp/python/3.5.2/Python-3.5.2.tgz"
converted 'http://www.python.org/ftp/python/3.5.2/Python-3.5.2.tgz' (ANSI_X3.4-1968) -> 'http://www.python.org/ftp/python/3.5.2/Python-3.5.2.tgz' (UTF-8)
--2016-10-19 09:09:43-- http://www.python.org/ftp/python/3.5.2/Python-3.5.2.tgz
Resolving www.python.org (www.python.org)... 151.101.88.223, 2a04:4e42:15::223
Connecting to www.python.org (www.python.org)|151.101.88.223|:80... connected.
HTTP request sent, awaiting response... 301 Moved Permanently
Location: https://www.python.org/ftp/python/3.5.2/Python-3.5.2.tgz [following]
converted 'https://www.python.org/ftp/python/3.5.2/Python-3.5.2.tgz' (ANSI_X3.4-1968) -> 'https://www.python.org/ftp/python/3.5.2/Python-3.5.2.tgz' (UTF-8)
--2016-10-19 09:09:43-- https://www.python.org/ftp/python/3.5.2/Python-3.5.2.tgz
Connecting to www.python.org (www.python.org)|151.101.88.223|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 20566643 (20M) [application/octet-stream]
Saving to: 'python.tgz'

python.tgz      100%[=====>] 19.61M 52.9KB/s  in 5m 48s

2016-10-19 09:15:33 (57.6 KB/s) - 'python.tgz' saved [20566643/20566643]

root@d9507a3c654b:/# tar xzf python.tgz

```

接着，进入到 Python 的源代码目录，进行编译前的配置、编译及安装。

```

root@d9507a3c654b:/# cd Python-3.5.2/
root@d9507a3c654b:/Python-3.5.2# ./configure --enable-shared && make && make install
checking build system type... x86_64-unknown-linux-gnu
checking host system type... x86_64-unknown-linux-gnu
checking for --enable-universalsdk... no
checking for --with-universal-archs... no
checking MACHDEP... linux
checking for --without-gcc... no
checking for --with-icc... no
checking for gcc... gcc
checking whether the C compiler works... yes
checking for C compiler default output file name... a.out
checking for suffix of executables...
checking whether we are cross compiling... no
checking for suffix of object files... o
checking whether we are using the GNU C compiler...
...

```



要运行 Python 解析程序，还要把源码目录中编译出来的一个共享库加入系统共享库目录中，这样才能够保证 Python 解析程序正确地使用这个库。

```
root@d9507a3c654b:/Python-3.5.2# cp libpython3.5m.so.1.0 /usr/lib/
```

在上述操作都完成之后，就可以通过 python3 命令来使用 Python 的解析程序了。这里使用 python3 命令来查看 Python 的版本信息。

```
root@d9507a3c654b:/Python-3.5.2# python3 --version
Python 3.5.2
```

### 10.3.3 构建 Python 镜像

根据之前进行的在 Docker 容器中安装 Python 解析程序的实践，可以很轻松地写出构建 Python 镜像的 Dockerfile。

```
# Python
# VERSION 0.0.1

# 基础镜像
FROM debian:jessie
# 维护者信息
MAINTAINER You Ming <youming@funcuter.org>

# 安装 Python
RUN buildDeps="wget gcc make autoconf tcl-dev tk-dev ca-certificates" \
    && apt-get update && apt-get install -y --no-install-recommends $buildDeps tcl tk \
# 下载 Python
    && wget -O python.tgz "http://www.python.org/ftp/python/3.5.2/Python-3.5.2.tgz" \
    && mkdir -p /usr/src/python \
    && tar -zxf python.tgz -C /usr/src/python --strip-components=1 \
    && cd /usr/src/python \
# 编译和安装
    && ./configure --enable-shared \
    && make \
    && make install \
    && cp libpython3.5m.so.1.0 /usr/lib/ \
    && rm -f python.tgz \
    && rm -rf /usr/src/python \
    && apt-get remove -y $buildDeps

# 打印 Python 的版本信息
```

```
CMD ["python3", "--version"]
```

将编写好的 Dockerfile 保存到文件中，再通过 docker build 命令构建 Python 镜像。

```
$ sudo docker build -t ymdot/python:0.0.1 ./python
Sending build context to Docker daemon 2.56 kB
Step 1 : FROM debian:jessie
----> 1b088884749b
Step 2 : MAINTAINER You Ming <youming@funcuter.org>
----> Using cache
----> 0b647e0ed3ee
Step 3 : RUN buildDeps="wget gcc make autoconf tcl-dev tk-dev"      && apt-get update
&& apt-get install -y --no-install-recommends $buildDeps tcl tk      && wget -O
python.tgz "http://www.python.org/ftp/python/3.5.2/Python-3.5.2.tgz"      && mkdir -p /
usr/src/python
----> Running in 02e5f08330ba
...
----> 59e1d83f2f57
Removing intermediate container 02e5f08330ba
Step 4 : CMD python3 --version
----> Running in fda4d727ab5b
----> a7545efe623c
Removing intermediate container fda4d727ab5b
Successfully built a7545efe623c
```

镜像构建完成后，可以从本地的镜像仓库中找到它。

```
$ sudo docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
ymdot/python	0.0.1	a7545efe623c	5 minutes ago	498.1 MB
...				

### 10.3.4 测试 Python 容器

因为在 Python 镜像中定义基于 Python 镜像的容器默认的启动命令为 python3 --version，所以如果直接基于这个镜像创建和运行容器，就可以在终端中看到程序打印出的 Python 版本信息。

```
$ sudo docker run --rm -t --name python ymdot/python:0.0.1
Python 3.5.2
```

要测试 Python 解析程序的运行状况，要先编写一个简单的 Python 程序。这里编写一个基于 Python 的逐渐显示时间的程序，大致的代码如下。

```
import time

while (1):
    print(time.ctime())
    time.sleep(1)
```

将 Python 程序保存为 `time.py`，然后创建基于 Python 镜像的容器，将这个文件以数据卷的方式挂载到容器中。对于这个容器的启动命令，我们使用运行这个 Python 程序代码文件的命令。

```
$ sudo docker run --rm -t --name python -v /ymdot/time.py:/time.py ymdot/python:0.0.1
python3 /time.py
Thu Oct 20 05:07:19 2016
Thu Oct 20 05:07:20 2016
Thu Oct 20 05:07:21 2016
Thu Oct 20 05:07:22 2016
Thu Oct 20 05:07:23 2016
...
```

当时间逐渐从屏幕上打印出来时，就表明 Python 程序已经正常运行了，这也表示基于前面构建的 Python 镜像所创建的容器能够正常地运行 Python 程序，即我们构建的 Python 镜像的实践是成功的。

## 10.4 Node.js

Node.js 是一个创造性地将 JavaScript 脚本语言从浏览器带入更广阔领域的工具，它可以把 JavaScript 作为服务器动态解析语言来处理 Web 请求。在 Docker 容器里，我们也能使用 Node.js 来处理 Web 请求。

### 10.4.1 Node.js 简介

Node.js 是一个跨平台的脚本语言运行环境，运行以 JavaScript 写成的程序代码，并能在市面上几乎所有的操作系统中运行。

Node.js 基于 Google 开发的 V8 引擎，是一个能够将 JavaScript 代码编译为机器代码的 JavaScript 运行环境。Google 开发 V8 引擎的主要目的是提升 Google Chrome，即 Google 浏览器中 JavaScript 的运行速度。2009 年，Ryan Dahl 等人巧妙地将 V8 引擎用于运行浏



览器之外的 JavaScript 代码，并很快得到了众多开发者的赞赏。随后，Node.js 应运而生。

通过 V8 引擎对运行速度的加持，Node.js 让 JavaScript 开始摆脱浏览器的束缚，演进到更广阔的开发空间中。也正是由于 Node.js 的推动，JavaScript 由小众的前端语言变成了可以独立运行并处理任务的大众语言。可以说，Node.js 把 JavaScript 的简单易用和程序跨平台支持相结合，创造了新的 JavaScript 世界。图 10-4 展示了 Node.js 的标识。

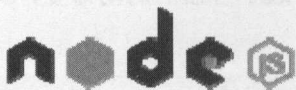


图 10-4 Node.js 的标识

Node.js 的一大特点是直接集成和提供了事件驱动和非阻塞 IO，这些设计弥补了 Node.js 以单线程运行 JavaScript 代码的缺陷。特别是在处理高并发请求时，虽然 Node.js 只能以单线程处理，但是处理代码可以以回调函数的形式插入异步 IO 中。加之事件机制能够非常灵活地切换处理状态，这就让 Node.js 在应对大并发事件上依然可以得心应手。

Node.js 提供了文件 IO、网络操作、数据流、加密算法等模块，使开发者能够直接通过这些模块进行相应的操作。同时，得益于 NPM (Node Package Manager, Node.js 包管理器)，我们可以很容易地引入和使用其他开发者开发的 Node.js 项目，这可以使程序运行更快，也更能体现基于互联网的协作精神。

## 10.4.2 安装 Node.js

在实践 Node.js 在 Docker 容器中的安装过程之前，需要先创建一个用于进行安装实践的 Docker 容器。这里选择 Debian 系统镜像作为容器的基础镜像。首先创建一个基于 Debian 系统镜像的容器，并进入到容器中。

```
$ sudo docker run -it --rm --name nodejs debian:jessie /bin/bash
root@41aa7d7abc1e:/#
```

Node.js 官方提供了直接运行的程序，所以不需要对程序进行编译安装，只需要从 Node.js 官网上将程序下载到本地运行即可。

由于 Docker 的 Debian 镜像是一个精简镜像，所以容器中没有用于下载的工具，需要从 Debian 的 APT 软件仓库中安装 wget，用于 Node.js 的下载。在安装 wget 之前，需要更新 APT 软件仓库中的软件源。

```
root@41aa7d7abc1e:/# apt-get update
Get:1 http://security.debian.org jessie/updates InRelease [63.1 kB]
```

```

Ign http://httpredir.debian.org jessie InRelease
Get:2 http://httpredir.debian.org jessie-updates InRelease [142 kB]
Get:3 http://httpredir.debian.org jessie-updates/main amd64 Packages [17.6 kB]
Get:4 http://httpredir.debian.org jessie Release.gpg [2373 B]
Get:5 http://httpredir.debian.org jessie Release [148 kB]
Get:6 http://httpredir.debian.org jessie/main amd64 Packages [9064 kB]
Get:7 http://security.debian.org jessie/updates/main amd64 Packages [391 kB]
Fetched 9829 kB in 19s (515 kB/s)
Reading package lists... Done

```

更新软件源之后，安装执行 `wget`。

```

root@41aa7d7abc1e:/# apt-get install -y wget
Reading package lists... Done
Building dependency tree... Done
The following extra packages will be installed:
  ca-certificates libffi6 libgmp10 libgnutls-deb0-28 libhogweed2 libicu52 libidn11
  libnettle4
  libp11-kit0 libpsl0 libssl1.0.0 libtasn1-6 openssl
Suggested packages:
  gnutls-bin
The following NEW packages will be installed:
  ca-certificates libffi6 libgmp10 libgnutls-deb0-28 libhogweed2 libicu52 libidn11
  libnettle4
  libp11-kit0 libpsl0 libssl1.0.0 libtasn1-6 openssl wget
0 upgraded, 14 newly installed, 0 to remove and 18 not upgraded.
Need to get 10.8 MB of archives.
After this operation, 38.9 MB of additional disk space will be used.
Do you want to continue? [Y/n] y^C
root@41aa7d7abc1e:/# apt-get install -y wget
Reading package lists... Done
Building dependency tree... Done
The following extra packages will be installed:
  ca-certificates libffi6 libgmp10 libgnutls-deb0-28 libhogweed2 libicu52 libidn11
  libnettle4
  libp11-kit0 libpsl0 libssl1.0.0 libtasn1-6 openssl
Suggested packages:
  gnutls-bin
The following NEW packages will be installed:
  ca-certificates libffi6 libgmp10 libgnutls-deb0-28 libhogweed2 libicu52 libidn11
  libnettle4
  libp11-kit0 libpsl0 libssl1.0.0 libtasn1-6 openssl wget
0 upgraded, 14 newly installed, 0 to remove and 18 not upgraded.

```

```
Need to get 10.8 MB of archives.
After this operation, 38.9 MB of additional disk space will be used.
Get:1 http://security.debian.org/ jessie/updates/main libssl1.0.0 amd64 1.0.1t-
1+deb8u5 [1048 kB]
...
Running hooks in /etc/ca-certificates/update.d....done.
```

接着，使用 `wget` 命令从 Node.js 官网中下载 Node.js。目前 Node.js 分为两个主线版本，4.6.0 版本为长期支持的版本，这里就使用 4.6.0 版本的 Node.js。

```
root@41aa7d7abc1e:/# wget -O node.tar.gz "https://nodejs.org/dist/v4.6.0/node-
v4.6.0-linux-x64.tar.gz" converted 'https://nodejs.org/dist/v4.6.0/node-v4.6.0-
linux-x64.tar.gz' (ANSI_X3.4-1968) -> 'https://nodejs.org/dist/v4.6.0/node-v4.6.0-
linux-x64.tar.gz' (UTF-8)
--2016-10-17 06:03:47-- https://nodejs.org/dist/v4.6.0/node-v4.6.0-linux-x64.tar.gz
Resolving nodejs.org (nodejs.org)... 104.20.22.46, 104.20.23.46, 2400:cb00:2048:1::
6814:162e, ...
Connecting to nodejs.org (nodejs.org)|104.20.22.46|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 12148698 (12M) [application/gzip]
Saving to: 'node.tar.gz'
```

由于 Node.js 包中就是可执行的程序，所以不需要再进行编译等操作，可以直接把程序包解压到存放程序的目录中。

```
root@41aa7d7abc1e:/# tar -xzf "node.tar.gz" -C /usr/local --strip-components=1
```

至此，安装 Node.js 的操作就结束了，通过 `node` 命令就可以使用 Node.js 了。这里通过 `node` 命令查看 Node.js 的版本，检查 Node.js 是否能够正常运行。

```
root@41aa7d7abc1e:/# node -v
v4.6.0
```

### 10.4.3 构建 Node.js 镜像

根据之前在 Docker 容器中搭建 Node.js 程序的实践，可以把构建 Node.js 镜像的过程写成对应的 Dockerfile。

```
# Node.js
# VERSION 0.0.1

# 基础镜像
FROM debian:jessie
```



```

# 维护者信息
MAINTAINER You Ming <youming@funcuter.org>

# 安装 Node.js
RUN buildDeps="wget" \
    && apt-get update && apt-get install -y $buildDeps --no-install-recommends \
# 下载 Node.js
    && wget -O node.tar.gz "http://nodejs.org/dist/v4.6.0/node-v4.6.0-linux-x64.
tar.gz" \
    && tar -zxf "node.tar.gz" -C /usr/local --strip-components=1 \
    && rm node.tar.gz \
    && apt-get remove -y $buildDeps

# 打印 Node.js 的版本信息
CMD ["node", "-v"]

```

将 Dockerfile 保存为文件，并使用 `docker build` 命令将 Dockerfile 构建为 Node.js 镜像。

```

$ sudo docker build -t ymdot/nodejs:0.0.1 ./nodejs
Sending build context to Docker daemon 2.56 kB
Step 1 : FROM debian:jessie
----> 1b088884749b
Step 2 : MAINTAINER You Ming <youming@funcuter.org>
----> Using cache
----> 0b647e0ed3ee
Step 3 : RUN buildDeps="wget" && apt-get update && apt-get install -y $buildDeps
--no-install-recommends && wget -O node.tar.xz "https://nodejs.org/dist/v$NODE_
VERSION/node-v4.6.0-linux-x64.tar.xz"
    && tar -zxf "node.tar.gz" -C /usr/local --strip-components=1 && rm node.tar.gz
&& apt-get remove -y $buildDeps
----> Running in 2169a6fb9092
...
----> 5ealcfa140fb
Removing intermediate container 3382229e63bf
Step 4 : CMD node -v
----> Running in c2eb94cd2c71
----> fd86bc665d84
Removing intermediate container c2eb94cd2c71
Successfully built fd86bc665d84

```

镜像构建完成后，可以从本地镜像仓库中找到它。

```
$ sudo docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
ymdot/nodejs	0.0.1	fd86bc665d84	14 minutes ago	204.6 MB
...				

## 10.4.4 测试 Node.js 容器

如果直接通过之前构建的 Node.js 镜像创建容器，可以看到终端中打印出了 Node.js 的版本信息。因为在构建镜像的 Dockerfile 中，定义容器的默认启动命令就是查看 Node.js 版本的命令。

```
$ sudo docker run -t --rm --name nodejs ymdot/nodejs:0.0.1
v4.6.0
```

如果要运行一个简单的 JavaScript 程序，判断 Node.js 镜像是否能够提供 JavaScript 程序运行的环境，需要先编写一个简单的能够运行在 Node.js 中的 JavaScript 程序。这里仍然使用打印当前时间这个目标进行 JavaScript 程序的编写，程序的简单代码如下。

```
function showTime() {
    var date = new Date();
    console.log(date.toString());
}

setInterval(showTime, 1000);
```

将这段代码保存为 time.js，并通过 Node.js 镜像创建新的容器，将这个代码文件通过数据卷的形式挂载到容器中。对于启动容器的命令，我们使用 node 程序来运行这个 JavaScript 代码文件。

```
$ sudo docker run -t --name nodejs -v /ymdot/time.js:/time.js ymdot/nodejs:0.0.1 node /
time.js
Tue Oct 18 2016 07:21:21 GMT+0000 (UTC)
Tue Oct 18 2016 07:21:22 GMT+0000 (UTC)
Tue Oct 18 2016 07:21:23 GMT+0000 (UTC)
Tue Oct 18 2016 07:21:24 GMT+0000 (UTC)
Tue Oct 18 2016 07:21:25 GMT+0000 (UTC)
...
```

当我们看到时间以每秒一行的速度显示到终端时，表示 JavaScript 程序能够在容器的 Node.js 中执行，也说明构建的 Node.js 镜像是可以正常使用的。

## 10.5 本章小结

动态 Web 页面的生成在很大程度上依赖动态处理程序，在本章中，我们就对目前常用的动态处理程序在 Docker 中的搭建进行了讲解和实践。通过本章的学习，我们分别掌握了构建 Java、PHP、Python、Node.js 镜像的步骤，也对如何使用这些镜像来创建和运行能够处理动态 Web 请求的容器的过程有了认识。

掌握了动态处理程序之后，我们对搭建一个简单的能够处理用户请求、保存用户数据并生成不同页面的 Web 服务器所需的程序，以及它们在 Docker 中的使用有了了解，也能够使用 Docker 搭建和部署一个完整的 Web 服务器了。



# 第 11 章

## 综合演练

在之前的章节实践中，我们分别学习了在 Docker 中搭建和使用 Web 服务器程序、数据库、缓存工具及动态处理程序。本章我们就对这些工具进行组合，通过 Docker 搭建一个完整的 Web 项目。

### 11.1 演练目标

在进行实践前，需要设定一个实践目标，本次实践的目标是搭建一个完整的 Web 服务器。

#### 11.1.1 目标概述

完整的 Web 服务器通常包含 Web 服务器程序、数据库、缓存及动态处理程序。这几项程序在之前的章节中已经提到过，并分别进行了在 Docker 中搭建和部署的实践。在这次实践中，就将这几项工具加以组合，进行在 Docker 中搭建完整软件体系的练习。

在这次实践中，我们设定了以下几个主要目标：

- 在 Docker 中使用运行在多个容器中的不同程序，组合成一套完整的 Web 服务体系。

- 实现向数据库中写入数据，从数据库中读取数据并展示到网页中。
- 实现向缓存中写入数据，从缓存中读取数据并展示到网页中。
- 通过页面交互，实现数据库数据到缓存的更新。

结合实践的目标，我们采用 Nginx + Memcached + MySQL + PHP 的架构，用 Nginx 承担 Web 服务器程序的任务、用 Memcached 处理缓存、用 MySQL 存储数据、用 PHP 进行请求处理和逻辑判断。通过这四个软件的协调，完成整个 Web 服务的功能，并且达到设定的实践目标。

根据 Docker 的推荐形式，分别将 4 个程序部署在 4 个不同的容器中。存在通信关系的程序，比如 PHP 与 MySQL 之间的通信，采用容器间通信的方式来实现。在整个系统中，暴露 MySQL 容器的 3306 端口用于进行对数据库的操作，暴露 Nginx 容器的 80 端口用于提供 Web 服务。

## 11.1.2 代码编写

确定了方案之后，需要进行代码的编写，这里的代码主要指 PHP 程序中的代码。由于编写 PHP 程序的知识不属于本书的范畴，我们此处只展示代码，不做过多解读。

我们需要编写两个代码文件，一个用于网页的展示，向用户提供操作界面；另一个用于处理用户的请求，将输入传输到页面上。

其中，index.html 文件用于展示和提供用户操作入口，其代码如下：

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">

    <title>You Ming Docker Demo</title>

    <link rel="stylesheet" href="//cdn.bootcss.com/bootstrap/3.3.0/css/bootstrap.min.css">
    <script src="//cdn.bootcss.com/jquery/2.2.4/jquery.min.js"></script>
  </head>

  <body>
    <div class="container">
```

```

<div class="page-header">
  <h2>存储数据</h2>
  <div class="form-group">
    <label for="write-key">数据键名</label>
    <input type="text" class="form-control" id="write-key">
  </div>
  <div class="form-group">
    <label for="write-value">数据键值</label>
    <input type="text" class="form-control" id="write-value">
  </div>
  <button type="button" id="write" class="btn btn-default">存储</button>
</div>
<div class="page-header">
  <h2>同步数据</h2>
  <button type="button" id="sync" class="btn btn-default">同步</button>
</div>
<div class="page-header">
  <h2>读取数据</h2>
  <div class="form-group">
    <label for="read-key">数据键名</label>
    <input type="text" class="form-control" id="read-key">
  </div>
  <button type="button" id="read" class="btn btn-default">读取</button>
  <h4 id="read-result"></h4>
</div>
</body>

<script type="application/javascript">
  $('#write').click(function () {
    $.post(
      'http://localhost/action.php',
      {
        action: 'write',
        key: $('#write-key').val(),
        value: $('#write-value').val()
      },
      function () {
        $('#write-key').val('');
        $('#write-value').val('');
      }
    );
  });

```



```

$('#sync').click(function () {
    $.post(
        'http://localhost/action.php',
        {
            action: 'sync'
        }
    );
});
$('#read').click(function () {
    $.post(
        'http://localhost/action.php',
        {
            action: 'read',
            key: $('#read-key').val()
        },
        function (data) {
            $('#read-result').html(data);
        }
    );
});
</script>
</html>

```

action.php 文件用于处理用户的请求，其代码如下：

```

<?php

function ymdot_write($key, $value)
{
    $mysql = mysqli_connect('ymdot-mysql', 'root', 'screencast', 'web');
    $rs = mysqli_fetch_all(mysqli_query($mysql, "SELECT * FROM `ymdot` WHERE `key` = '$key'"), MYSQLI_ASSOC);
    if ($rs) {
        mysqli_query($mysql, "UPDATE `ymdot` SET `value` = '$value' WHERE `key` = '$key'");
    } else {
        mysqli_query($mysql, "INSERT INTO `ymdot` (`key`, `value`) VALUES ('$key', '$value')");
    }
    mysqli_close($mysql);
}

function ymdot_sync()

```

```

{
    $mysql = mysqli_connect('ymdot-mysql', 'root', 'screencast', 'web');
    $rs = mysqli_fetch_all(mysqli_query($mysql, "SELECT * FROM `ymdot`"), MYSQLI_
ASSOC);
    mysqli_close($mysql);

    $memcache = memcache_connect('ymdot-memcached', 11211);
    memcache_flush($memcache);
    foreach ($rs as $row) {
        memcache_set($memcache, $row['key'], $row['value']);
    }
    memcache_close($memcache);
}

function ymdot_read($key)
{
    $mysql = mysqli_connect('ymdot-mysql', 'root', 'screencast', 'web');
    $rs = mysqli_fetch_all(mysqli_query($mysql, "SELECT * FROM `ymdot` WHERE `key` =
'$key'"), MYSQLI_ASSOC);
    $row = reset($rs);
    $dbVal = isset($row['value']) ? $row['value'] : '';

    $memcache = memcache_connect('ymdot-memcached', 11211);
    $memVal = memcache_get($memcache, $key);
    memcache_close($memcache);

    echo '数据库中的值: ' . $dbVal . '<br/>' . '缓存中的值: ' . $memVal;
}

$action = $_POST['action'];

switch ($action) {
    case 'write':
        ymdot_write($_POST['key'], $_POST['value']);
        break;
    case 'sync':
        ymdot_sync();
        break;
    case 'read':
        ymdot_read($_POST['key']);
        break;
}

```

## 11.2 环境搭建

利用 Docker 可以让开发与生产运行得到一套一致的环境,为用户节约了大量的时间和精力。

### 11.2.1 准备镜像

在之前的几个章节中,学习和实践了我们所需要的 Web 服务器程序 Nginx、MySQL、Memcached、PHP 在 Docker 容器中的安装与配置;也对包含这些程序的 Docker 镜像进行了构建,并编写了能够构建这些镜像的 Dockerfile。在本章的实践中,我们用另外一种方式来获得这 4 个软件的 Docker 镜像,即从 Docker Hub 远程镜像仓库中获取这些镜像。

通过 Docker Hub 获得镜像,不但能够通过共享镜像仓库快速获得他人构建的镜像、减少构建和测试镜像的时间、提高搭建运行环境的速度;也能够有效地避免在搭建相关程序的过程中,因为不熟悉或者缺漏关键的配置,导致程序运行可能存在的问题。可以说,充分利用 Docker Hub 等 Docker 镜像仓库,避免自己再造轮子,并不是一种偷懒的表现。恰恰相反,这有助于我们熟练使用 Docker 来提升效率。

在了解了我们为什么选择从 Docker Hub 中获取本章所需的 Docker 镜像之后,就可以开始从 Docker Hub 中分别获取 Nginx、MySQL、Memcached 和 PHP 的镜像。

对于 Nginx 的镜像我们采用 stable 标签。目前这个标签所代表的 Nginx 镜像中,Nginx 的最新稳定版本是 1.10.2,比较适合我们使用。

```
$ sudo docker pull nginx:stable
stable: Pulling from library/nginx

43c265008fae: Pull complete
31ad8dd8d4c8: Pull complete
9de865e63723: Pull complete
Digest: sha256:7bdf45d2355a66eb31288d4c9b69b60e52c31b3934c7ffeb139356741e52a468
Status: Downloaded newer image for nginx:stable
```

对于 MySQL 镜像,我们采用标签为 5.6 的镜像。目前这个标签的 MySQL 镜像内包含的是 5.6.34 版本的 MySQL 程序。

```
$ sudo docker pull mysql:5.6
```



```
5.6: Pulling from library/mysql
```

```
43c265008fae: Pull complete
d7abd54d3b34: Pull complete
92b527830a1b: Pull complete
44839710d611: Pull complete
3828a16bed5c: Pull complete
fb91763f6b4e: Pull complete
892bfb27c685: Pull complete
02874ec7a2dc: Pull complete
861c1296cc0d: Pull complete
d611998d5598: Pull complete
09037dc5a941: Pull complete
Digest: sha256:b714e9bd05f38877832a976813dec03650c3f4a4b09cd23652a85efd41ae6cd8
Status: Downloaded newer image for mysql:5.6
```

对于 Memcached，我们选择标签为 1.4 的 Memcached 镜像。目前这个标签所对应的 Memcached 镜像中 Memcached 的版本为 1.4.32。

```
$ sudo docker pull memcached:1.4
```

```
1.4: Pulling from library/memcached
```

```
43c265008fae: Already exists
58b25e9455e9: Pull complete
add2ca9ff2c6: Pull complete
c2e984816f0d: Pull complete
f1f468397f92: Pull complete
ffcl1e1b5d115: Pull complete
Digest: sha256:c227df5a7c824dd054ede097962b3cb347575aeb97caf3ef6c34e581cd6d31fa
Status: Downloaded newer image for memcached:1.4
```

大家也许会问，为什么选择程序的 Docker 镜像时都喜欢使用 5.6 或 1.4 这类两级版本号 的镜像，而程序本身就拥有三级版本号，并且也存在对应的三级版本号作为标签的镜像。

这主要是在通用的程序三级版本命名里，版本号的第二部分，也就是次级版本号代表程序功能上的更迭，也是能够确定程序功能范围的版本号。而最后一级版本号只用于 Bug 修复等细节上的补充。由于程序 Bug 修改的速度可能非常快，如果采用确定三级版本的镜像标签，很有可能不能及时地跟进程序对这一功能性版本的修复性迭代。所以，通常采用程序的次级版本号来界定程序，也使用前两级版本号来确定程序对应容器所采用的标签。

我们需要特别注意 PHP 程序的镜像。在我们之前的实践中，采用 Docker 搭建了一个能够运行 PHP 程序的镜像。但 PHP 的运行方式有很多种，我们之前搭建的是基于终端运行的 PHP 程序的 Docker 镜像。在与 Nginx 搭配处理 Web 服务器请求的程序中，我们通常采用 PHP-FPM 程序进行反向代理对接。所以需要从远程仓库获取的是 PHP 镜像的 PHP-FPM 版本，我们选择了比较常用的 5.6 版本的 PHP。在 PHP 镜像的 5.6 标签下，目前对应的 PHP 版本为 5.6.27。而带有 PHP-FPM 支持的 5.6 版本的 PHP 镜像，对应的镜像标签为 5.6-fpm，所以通过这个标签获得 PHP 镜像。

```
$ sudo docker pull php:5.6-fpm
5.6-fpm: Pulling from library/php

43c265008fae: Already exists
6ee27d07994b: Pull complete
d43536f442a0: Pull complete
caa40caf97a5: Pull complete
064a3d3e9245: Pull complete
bb8cd631ee52: Pull complete
3181f198b31f: Pull complete
32f4db8284a3: Pull complete
7ca292aae86f: Pull complete
Digest: sha256:d939b4647903dd60ed617c5ccb0755d7ce5a64e27b182454024a190f937b9ebd
Status: Downloaded newer image for php:5.6-fpm
```

为了确保之后的实践可以顺利进行，要保证包含我们所需的这四个工具的镜像已经下载到了本地镜像仓库中。特别是 Docker Hub 服务器架设在国外，时常会因网络不稳定造成镜像下载中断，此时就需要重新进行尝试。

我们可以通过本地镜像列表，查看这四个镜像是否已经下载到了本地镜像仓库中。

```
$ sudo docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
php	5.6-fpm	2600a6734827	7 days ago	361.1 MB
nginx	stable	9bd6b3c63114	7 days ago	180.7 MB
mysql	5.6	9ea46e46d5b3	7 days ago	326.8 MB
memcached	1.4	f410ea4110a3	7 days ago	126.1 MB

## 11.2.2 程序配置

要让 Web 服务正常运行，需要进行正确的配置。在需要搭建的由 Nginx、Memcached、MySQL 和 PHP 组成的 Web 服务体系中，需重点对 Nginx 和 PHP 程序进行配置。另外，

MySQL 其实也需要进行配置, 不过 Docker Hub 所提供的 MySQL 镜像已经通过 Entrypoint 脚本完成了对 MySQL 的配置, 所以不需要再以配置文件的形式进行定义, 只需在启动 MySQL 时通过环境变量传入根用户的密码即可。

Nginx 程序的配置位于 `/etc/nginx/nginx.conf` 中, 包含了对 Nginx 管理的网站的定义及有关请求的参数定义, 其内容如下。

```
user nginx;

worker_processes auto;
worker_rlimit_nofile 65535;

pid /var/run/nginx.pid;

events {
    worker_connections 2048;
    multi_accept on;
    use epoll;
}

http {
    include mime.types;
    default_type text/html;

    log_format main '$remote_addr - $remote_user [$time_local] "$request" '
        '$status $body_bytes_sent "$http_referer" '
        '"$http_user_agent" "$http_x_forwarded_for"';

    sendfile on;

    keepalive_timeout 60;

    client_header_timeout 15;
    client_body_timeout 15;
    send_timeout 30;

    charset UTF-8;

    gzip on;
    gzip_disable "msie6";
    gzip_proxied any;
    gzip_min_length 10k;
```



```
gzip_comp_level 4;
gzip_buffers 8 48k;
gzip_types text/plain text/css application/json application/x-javascript text/xml
application/xml application/xml+rss text/javascript;

server_tokens on;

client_max_body_size 100m;

fastcgi_connect_timeout 120;
fastcgi_send_timeout 300;
fastcgi_read_timeout 300;
fastcgi_buffers 16 128k;
fastcgi_buffer_size 512k;
fastcgi_busy_buffers_size 512k;
fastcgi_temp_file_write_size 1024k;

server {
    listen 80;
    server_name _;

    root /www;

    index index.php index.html index.htm;

    location / {
        try_files $uri $uri/ @rewrite;
    }

    location @rewrite {
        rewrite ^(.*)$ /index.php$1;
    }

    location ~ \.php {
        fastcgi_pass ymdot-php:9000;

        include fastcgi_params;

        fastcgi_split_path_info ^(.+?\.php)(.*)$;
        fastcgi_param PATH_INFO $fastcgi_path_info;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        fastcgi_param PATH_REANSLATED $document_root$fastcgi_path_info;
```

```
#fastcgi_param HTTPS on;
}
}
}
```

在 Nginx 配置中, `fastcgi_pass ymdot-php:9000` 表示 Nginx 与 PHP 容器进行衔接所使用的主机和端口信息。我们定义 `ymdot-php` 为 PHP 容器的名称, 9000 端口为 PHP-FPM 程序进行监听和连接的端口。

PHP 的配置文件有两个: 一个用于配置 PHP 程序, 位于 `/usr/local/etc/` `php.ini` 中; 一个用于配置 PHP-FPM, 位于 `/usr/local/etc/php-fpm.conf` 中。这两个文件的配置内容如下。

`php.ini`:

```
[PHP]
engine = On
short_open_tag = Off
asp_tags = Off
precision = 14
output_buffering = 4096
zlib.output_compression = Off
implicit_flush = Off
unserialize_callback_func =
serialize_precision = 17
disable_functions =
disable_classes =
zend.enable_gc = On
expose_php = On

max_execution_time = 30
max_input_time = 60

memory_limit = 128M

error_reporting = E_ALL & ~E_DEPRECATED & ~E_STRICT

display_errors = On

display_startup_errors = Off

log_errors = On
log_errors_max_len = 1024
```

```
ignore_repeated_errors = Off
ignore_repeated_source = Off

report_memleaks = On
track_errors = Off
html_errors = On

variables_order = "GPCS"
request_order = "GP"
register_argc_argv = Off
auto_globals_jit = On

post_max_size = 8M
auto_prepend_file =
auto_append_file =
default_mimetype = "text/html"

doc_root =
user_dir =

enable_dl = Off

file_uploads = On
upload_max_filesize = 20M
max_file_uploads = 20

allow_url_fopen = On
allow_url_include = Off

default_socket_timeout = 60

[CLI Server]
cli_server.color = On

[Pdo_mysql]
pdo_mysql.cache_size = 2000
pdo_mysql.default_socket=

[mail function]
SMTP = localhost
smtp_port = 25
mail.add_x_header = On
```



```
[SQL]
sql.safe_mode = Off

[ODBC]
odbc.allow_persistent = On
odbc.check_persistent = On
odbc.max_persistent = -1
odbc.max_links = -1
odbc.defaultlrl = 4096
odbc.defaultbinmode = 1

[Interbase]
ibase.allow_persistent = 1
ibase.max_persistent = -1
ibase.max_links = -1
ibase.timestampformat = "%Y-%m-%d %H:%M:%S"
ibase.dateformat = "%Y-%m-%d"
ibase.timeformat = "%H:%M:%S"

[MySQL]
mysql.allow_local_infile = On
mysql.allow_persistent = On
mysql.cache_size = 2000
mysql.max_persistent = -1
mysql.max_links = -1
mysql.default_port =
mysql.default_socket =
mysql.default_host =
mysql.default_user =
mysql.default_password =
mysql.connect_timeout = 60
mysql.trace_mode = Off

[MySQLi]
mysqli.max_persistent = -1
mysqli.allow_persistent = On
mysqli.max_links = -1
mysqli.cache_size = 2000
mysqli.default_port = 3306
mysqli.default_socket =
mysqli.default_host =
```

```
mysqli.default_user =  
mysqli.default_pw =  
mysqli.reconnect = Off  
  
[mysqlnd]  
mysqlnd.collect_statistics = On  
mysqlnd.collect_memory_statistics = Off  
  
[PostgreSQL]  
pgsql.allow_persistent = On  
pgsql.auto_reset_persistent = Off  
pgsql.max_persistent = -1  
pgsql.max_links = -1  
pgsql.ignore_notice = 0  
pgsql.log_notice = 0  
  
[Sybase-CT]  
sybct.allow_persistent = On  
sybct.max_persistent = -1  
sybct.max_links = -1  
sybct.min_server_severity = 10  
sybct.min_client_severity = 10  
  
[bcmath]  
bcmath.scale = 0  
  
[Session]  
session.save_handler = files  
session.use_cookies = 1  
session.use_only_cookies = 1  
session.name = PHPSESSID  
session.auto_start = 0  
session.cookie_lifetime = 0  
session.cookie_path = /  
session.cookie_domain =  
session.cookie_httponly =  
session.serialize_handler = php  
session.gc_probability = 1  
session.gc_divisor = 1000  
session.gc_maxlifetime = 1440  
session.referer_check =  
session.cache_limiter = nocache
```



```
session.cache_expire = 180
session.use_trans_sid = 0
session.hash_function = 0
session.hash_bits_per_character = 5
url_rewriter.tags = "a:href,area:href,frame:src,input:src,form:fakeentry"

[MSSQL]
mssql.allow_persistent = On
mssql.max_persistent = -1
mssql.max_links = -1
mssql.min_error_severity = 10
mssql.min_message_severity = 10
mssql.compatability_mode = Off
mssql.secure_connection = Off

[Tidy]
tidy.clean_output = Off

[soap]
soap.wsdl_cache_enabled=1
soap.wsdl_cache_dir="/tmp"
soap.wsdl_cache_ttl=86400
soap.wsdl_cache_limit = 5

[ldap]
ldap.max_links = -1
```

### php-fpm.conf:

```
[www]
user = www-data
group = www-data

listen = 0.0.0.0:9000
listen.backlog = -1

pm = dynamic
pm.max_children = 500
pm.start_servers = 40
pm.min_spare_servers = 20
pm.max_spare_servers = 60
pm.process_idle_timeout = 20s
pm.max_requests = 500
```



除了要对 PHP 进行配置, 还要准备好 PHP 对接 MySQL 和 Memcached 的工具类库。因为 PHP 是以扩展的形式松散地组装所需要的类库的, 因此需要分别获取用于对接 MySQL 的 `mysql.so` 和用于对接 Memcached 的 `memcache.so`。

这两个扩展我们都可以在 PHP 环境下通过编译来获得, 也可以利用 PHP 容器来完成这个操作。首先, 启动一个新的 PHP 容器, 挂载一个宿主机目录导出编译的 PHP 扩展。

```
$ sudo docker run -it --rm -v /ymdot/php/exts:/php/exts php:5.6-fpm bash
```

接着, 进行 Memcached 扩展的编译, 这里直接通过 PHP 自带的扩展管理程序进行。

```
root@672b59e122b9:/usr/src# pecl install memcache
downloading memcache-2.2.7.tgz ...
Starting to download memcache-2.2.7.tgz (36,459 bytes)
.....done: 36,459 bytes
11 source files, building
running: phpize
Configuring for:
PHP Api Version:      20131106
Zend Module Api No:   20131226
Zend Extension Api No: 220131226
...
Build process completed successfully
Installing '/usr/local/lib/php/extensions/no-debug-non-zts-20131226/memcache.so'
install ok: channel://pecl.php.net/memcache-2.2.7
configuration option "php_ini" is not set to php.ini location
You should add "extension=memcache.so" to php.ini
```

之后, 进行 MySQL 扩展的编译, 在 PHP 源码目录中找到包含 MySQL 扩展源码的目录, 对其进行编译即可。

```
root@672b59e122b9:/usr/src/php-5.6.27/ext/mysql# phpize && ./configure && make &&
make install
Configuring for:
PHP Api Version:      20131106
Zend Module Api No:   20131226
Zend Extension Api No: 220131226
checking for grep that handles long lines and -e... /bin/grep
checking for egrep... /bin/grep -E
...
Installing shared extensions:      /usr/local/lib/php/extensions/no-debug-non-zts-
```

```
20131226/
```

```
Installing header files: /usr/local/include/php/
```

完成两个扩展的编译之后，将两个扩展复制到挂载的导出目录，也就是将它们转移到宿主机中，以便之后使用。

```
root@672b59e122b9:/usr/src# cp /usr/local/lib/php/extensions/no-debug-non-zts-20131226/memcache.so /php/exts
root@672b59e122b9:/usr/src# cp /usr/local/lib/php/extensions/no-debug-non-zts-20131226/mysqli.so /php/exts
```

除此之外，还需要在 `php.ini` 配置文件中加上对这两个扩展的引用，提醒 PHP 程序加载这两个扩展。

```
extension = memcache.so
extension = mysqli.so
```

## 11.3 项目运行

下面将实践由多个容器组成的项目，不但要熟悉如何使用启动容器的命令，还要注意容器之间的依赖关系。

### 11.3.1 启动容器

在启动组成项目的各个容器时，要根据依赖关系进行启动。我们先启动 Memcached 和 MySQL，因为它们不需要依赖其他容器。

启动 Memcached 的命令相对简单，不需要进行过多配置。

```
$ docker run -d --name ymdot-memcached memcached:1.4
48d833cb72df97304813526fae466063bc8d995c5365f42f9d44db6f21285909
```

在启动 MySQL 容器时，需要通过环境变量 `MYSQL_ROOT_PASSWORD` 来设置 MySQL，这样才能对 MySQL 进行正确连接。

```
$ sudo docker run -d --name ymdot-mysql -p 3306:3306 -e MYSQL_ROOT_PASSWORD=screencast
mysql:5.6
dc35acccc8a37a089770c54c5c51f71124a88ce1e9d5c6777ac143eae5795b62
```

然后启动 PHP 容器，将用于 PHP 配置的 `php.ini` 及 `php-fpm.conf` 两个配置文件分别

挂载到对应目录下。同时,把存放代码的目录挂载到容器中,代码挂载的位置是/www。另外,要让 PHP 容器连接到 Memcached 容器和 MySQL 容器,以便 PHP 程序能够访问和操作这两个容器中的 Memcached 和 MySQL 程序,通过它们写入和获取数据。所以,这里通过--link 参数打开 PHP 与这两个容器间的网络通信通道。

```
$ sudo docker run -d --name ymdot-php -v /ymdot/www:/www -v /ymdot/php/php.ini:/usr/local/etc/php/php.ini:ro -v /ymdot/php/php-fpm.conf:/usr/local/etc/php-fpm.conf:ro -v /ymdot/php/extends:/usr/local/lib/php/extensions/no-debug-non-zts-20131226/ --link ymdot-mysql --link ymdot-memcached php:5.6-fpm
599cef3cc0d64610eacad73e196a5754c2abfe4da1073142d769d80807c2a722
```

最后启动 Nginx 程序,让它挂载配置信息及 Web 程序目录,并连接到 PHP 容器中。同时,打开 Nginx 容器的 80 端口,并绑定到宿主机上,使我们能够访问到 Nginx 容器所提供的 Web 服务。

```
$ sudo docker run -d --name ymdot-nginx -v /ymdot/nginx/nginx.conf:/etc/nginx/nginx.conf -v gw:\docker\demo\www:/www --link ymdot-php -p 80:80 nginx:stable
759e9788adc278786aee589656a86ecf49664b8929132c2f685b044688012993
```

我们可以通过 docker ps 命令确保这四个容器都已经启动。

```
$ sudo docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
759e9788adc2	nginx:stable	"nginx -g 'daemon off'"	6 hours ago	Up 3 seconds
		0.0.0.0:80->80/tcp, 443/tcp		ymdot-nginx
599cef3cc0d6	php:5.6-fpm	"php-fpm"	6 hours ago	Up 3 minutes
		9000/tcp		ymdot-php
dc35acccc8a3	mysql:5.6	"docker-entrypoint.sh"	9 hours ago	Up 2 hours
		0.0.0.0:3306->3306/tcp		ymdot-mysql
48d833cb72df	memcached:1.4	"docker-entrypoint.sh"	10 hours ago	Up 4 hours
		11211/tcp		ymdot-memcached

## 11.3.2 测试项目

在进行项目测试之前,先对 MySQL 数据库中的数据进行初始化。如图 11-1 所示,首先通过数据库管理程序及之前设置的连接密码,连接到位于 Docker 容器中的 MySQL 上。





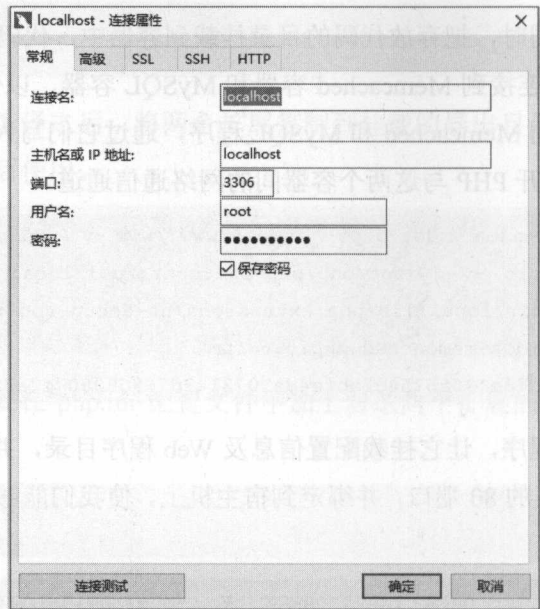


图 11-1 连接设置

如图 11-2 所示，在 MySQL 中新建一个数据库，用于程序测试数据的存储。

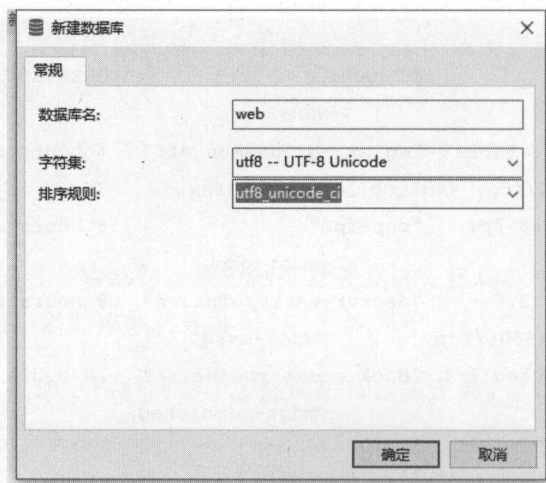


图 11-2 新建数据库

如图 11-3 所示，在新数据库中按设计新建一张数据表，并设置两个字段，分别用于存储数据的键名和键值。

名	类型	长度	小
key	varchar	255	0
value	varchar	255	0

图 11-3 新建数据表

完成准备工作之后就可以进行项目的测试了。如图 11-4 所示，先打开浏览器，进入 Web 服务提供的页面中。接着键入一条数据的名称与值，并单击“存储”按钮。



图 11-4 键入数据的名称与值

如图 11-5 所示，单击“读取”按钮，从数据库和缓存中读取存入的值。



图 11-5 读取存入的值

由于存入的值按照程序逻辑存入了数据库，而非缓存中，所以可以看到程序给出的结果存在于数据库中，在缓存中找不到相应的值。

如图 11-6 所示，单击“同步”按钮，将数据库中的数据同步到缓存中。

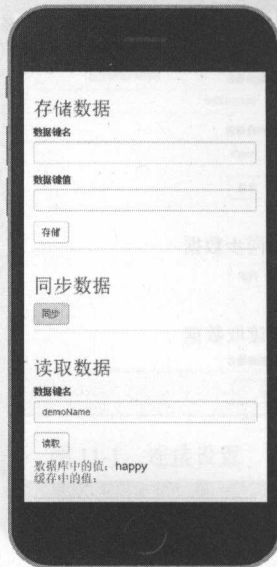


图 11-6 同步数据

再单击“读取”按钮，如图 11-7 所示，从数据库和缓存中读取键名对应的值。

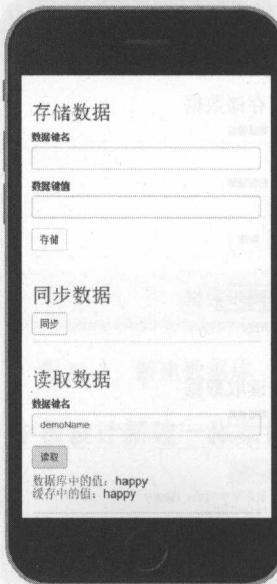


图 11-7 读取键名对应的值



可以看到，数据库和缓存中的值已经同步，都存储和缓存了键入的数据。

通过验证，我们所搭建的 Web 服务有了完整的请求处理、数据保存、缓存管理和页面展示的功能。这些功能分别由在 Docker 中所搭建的 Nginx 容器、MySQL 容器、Memcached 容器及 PHP 容器协同提供。这就表明，我们已经掌握了通过 Docker 组装和组合运行不同程序的多个容器，来完成我们对项目环境搭建的需求。同时也证明了 Docker 能够快速高效地为我们提供由多个模块组成的软件体系的运行环境。

## 11.4 本章小结

在本章中，我们通过一个由 Nginx、MySQL、Memcached 和 PHP 组成的完整 Web 服务体系，实践了如何通过 Docker 搭建和部署由多个不同程序模块所组成的系统。在实践中可以体会到 Docker 在部署效率方面的优势：只要记住容器的启动命令，就能马上在其他宿主机的 Docker 里部署一套完全一致的程序运行环境，再把代码和配置复制到对应的宿主机中，就能马上将项目运行起来。

使用好 Docker 的精髓并不在于多么熟练地通过它构建多么完美的镜像，而是如何利用它真正在开发和部署的过程中节约时间，提高生产效率。通过本章的实践，大家应该能够掌握如何将 Docker 运用到大型项目的部署上。

## 第三部分 提高篇

# 第 12 章

## 网络进阶

网络是计算机提供的重要资源，也是应用程序之间传递信息的主要方式之一。对于需要跨硬件限制、跨地域限制，包括跨出容器限制的场景，使用网络都是最佳的通信方式。在之前的章节中，我们对在 Docker 中进行网络的配置进行了简单介绍，了解了设置宿主机与容器的端口映射的方式，也讲解了实现容器间通信的方法。但与其他计算机资源不同，因为有着访问控制、协议栈、中继转发、路由、防火墙等众多实现方式、配置、属性，网络带有与生俱来的复杂性。所以在 Docker 中，为网络提供的配置和操作的方法还有很多。在本章里，我们就对之前没有提及的关于 Docker 网络方面的知识进行了解。

### 12.1 网络实现

在网络的发展过程中，我们建立了非常透明和统一的标准，而且随着硬件设备不断革新与发展，网络已经成为程序间进行沟通的重要方式。特别是在云计算领域，其存在众多计算机组成的集群，而这些主机间进行通信的方式大多以网络形式实现了。而 Docker 作为分布式部署的利器，自然少不了为容器内程序提供强大网络资源支持的模块。

### 12.1.1 容器网络基础

在 Docker 中，应用程序已经被隔离在了容器所定义的隔离范围之内。网络作为计算机资源的一部分，自然也会被容器进行隔离。但是，为了连接所有的微服务，组成完整的服务体系，大部分时候应用还需要与外界和其他容器中的程序进行通信。我们已经了解到，这种通信是利用网络来完成的。那么，Docker 是如何实现容器对网络的隔离的，又是如何在隔离的环境中实现数据通过网络传出或传入容器的呢？

我们知道，Docker 所提供的容器技术，是基于 Linux Kernel 提供的 Linux Container 的。在 Linux Container 中，包含能够用于隔离程序进程的 Namespaces。而在 Namespaces 技术里，存在一个用于隔离程序对网络信息调用的子模块，也就是 Network Namespaces。每个被 Network Namespaces 隔离的空间都拥有自己的网络设备、IP 地址、路由表、防火墙、网络配置、端口表等。Docker 正是借助 Network Namespaces 完成了对容器网络的隔离。

被隔离在 Network Namespaces 中的程序，是无法了解其他 Network Namespaces 或宿主机的网络环境与配置的。网络是双向的，既然不能了解对方的信息，就不能建立网络的连接。所以，通过 Network Namespaces，只具备了隔离网络的功能，还没有完成与其他网络连接的任务。

要突破 Network Namespaces 的隔离，可以使用 Veth Pair (Virtual Ethernet Pair) 来实现。Veth Pair 是一个虚拟网络通道，功能十分简单，就是将来自通道一端的数据输送到另外一端。这个简单的功能，正好可以帮助我们打穿被 Network Namespaces 隔离的网络，让内外网络能够通信。如图 12-1 所示，如果把 Network Namespace 比作我国西南地区的重重大山，那么运行在其中的程序就是大山深处的村寨，而 Veth Pair 则是一条隧道，打穿了整个山体，让山里面的人能够走出来，山外的人也能够走进去。

Veth Pair 打通了 Network Namespaces，实现了容器所隔离的网络环境与容器外部的网络通信。但 Veth Pair 只提供了两个端点，所以只能连接某两个网络终端。对于可能运行着众多容器的宿主机来说，如果只使用 Veth Pair 需要建立非常冗杂的数据隧道。因此，Docker 利用 Linux Bridge 把 Veth Pair 的一端都连接到宿主机中某个网桥所构成的交换机中，把容器连接不同的外部网络的任务由 Veth Pair 转移到了网桥上。这样，每个容器只需要建立一个 Veth Pair，就能连接任意外部的可达网络。并且，由于多个容器都连接到了同一个网桥，所以可以很容易地利用网桥构建的子网实现容器间的网络访问。





图 12-1 信息隧道

利用 Linux Bridge，不但不用为容器设置众多复杂的 Veth Pair 通信通道，也很容易地实现了容器间的网络连接。但用于连接容器的网桥，只是宿主机系统中虚拟生成的，与宿主机真实的网卡仍然不能互通。所以要实现容器与宿主机以外的网络互通，需要借助其他技术。在 Docker 的实现中，使用 Iptables 来解决这个问题。

Iptables 是 Linux 中用于管理网络过滤的程序，能够根据指定的规则，对位于 Linux 内核中的 netfilter 进行操作，实现对网络信息包过滤的功能。利用 Iptables，还可以实现端口的映射，让宿主机外部访问到容器的端口正是通过端口映射实现的。除了能够实现容器与宿主机外部网络进行沟通，Iptables 还能起到防火墙的作用，可以增强 Docker 网络的安全性。图 12-2 展示了容器与宿主机的网络关系。

所以，Docker 网络主要是由 Network Namespace、Veth Pair、Linux Bridge、Iptables 等技术实现的。

- ❑ Network Namespace: 实现了网络资源的隔离。对隔离环境提供了网络设备、协议栈、路由表、防火墙、/proc/net 目录、/sys/class/net 目录、端口表等网络配置和实现。
- ❑ Veth Pair: 实现了打穿隔离环境的网络传输数据通道。在 Docker 中，它的一端连接到容器中虚拟的网卡上，另一端连接到宿主机中专用的网桥上，通过这种方式实现了 Docker 容器与外部网络的互通。
- ❑ Linux Bridge: 放置在宿主机中的网桥，起到网络交换机的作用。因为容器网络通过 Veth Pair 连接到了网桥上，所以它能够在容器间转发网络数据。

- Iptables: 用于提供网络数据透传、NAT 等功能, 也可以利用它实现 Docker 网络的防火墙等网络安全防护的需求。

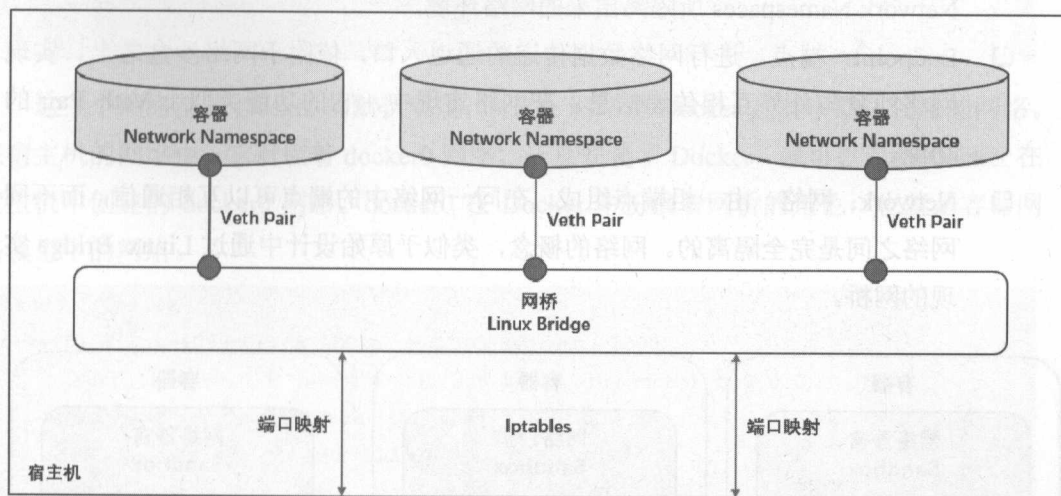


图 12-2 容器与宿主机的网络关系

## 12.1.2 网络模型

Docker 可以说是起于草莽的一款软件, 也可以说是站在巨人肩膀上设计出的一款软件。在最初的版本中, Docker 几乎就是其他开源软件或技术打的包而已。随着热衷 Docker 的开发者越来越多, 他们逐渐对 Docker 已有的结构进行优化、对基础架构进行整理和重新设计, 希望将 Docker 提升到新的高度。

Docker 团队将对容器实现的层进行了抽象, 形成了一套统一的容器支持对接接口, Libcontainer 就是 Docker 团队自己进行的一种实现。通过这种驱动模式的设计, Docker 可以更好地吸纳不同的实现, 让全世界的开发者在为其贡献力量的同时, 减少了不同环境之间的耦合, 提高了开发效率。同时, 这种方式也让我们能够拥有更多的选择, 根据实际情况选择合适的部件来组装 Docker, 让 Docker 得到更好的发挥。

Docker 的网络部分也在发展的过程中逐渐完善, 形成了 CNM (Container Network Model, 容器网络模型) 的配置, 如图 12-3 所示。Docker 将容器网络模型作为设计结构, 实现了 Libnetwork 库, 把原有的 Network Namespaces、Veth Pair、Linux Bridge、Iptables 等进行了封装。容器网络模型规范了 Docker 容器对网络使用的方式, 也让更多开发者可以根据容器网络模型所定义的结构和接口, 去实现自己需要的网络驱动。

容器网络模型将 Docker 内部网络分为了以下三个模块。

- ❑ **Sandbox:** 网络沙盒，也就是容器中隔离网络配置的虚拟环境，每个网络沙盒都拥有独立的网络配置等相关信息。在原有的网络实现中，网络沙盒类似于 Network Namespaces 所隔离出来的网络环境。
- ❑ **Endpoint:** 端点。进行网络数据传递的通道入口，依附于网络沙盒之上，实现网络沙盒与外界互相传递信息。在网络实现中，它的功能类似于 Veth Pair 的功能。
- ❑ **Network:** 网络。由一组端点组成，在同一网络中的端点可以互相通信，而不同网络之间是完全隔离的。网络的概念，类似于原始设计中通过 Linux Bridge 实现的网桥。

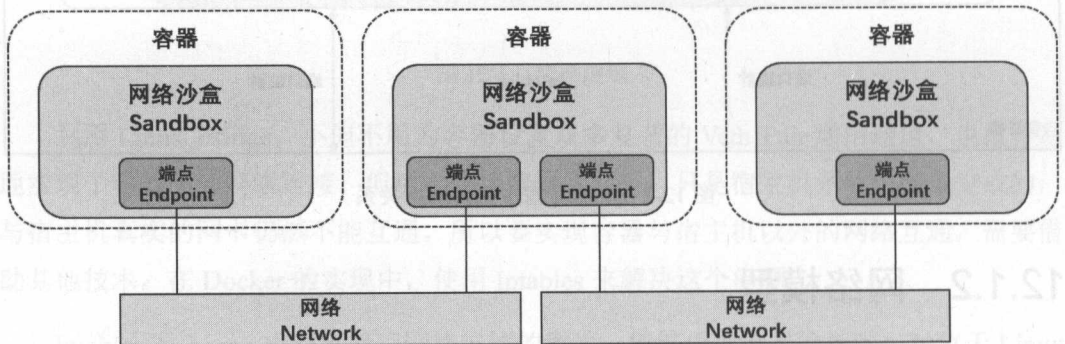


图 12-3 容器网络模型

在 Docker CLI 中，可以通过 `docker network` 命令对 Docker 的网络进行控制，包括其他形式的网络驱动，这就是通过容器网络模型统一的接口来实现的。

## 12.2 Docker 中的网络

Docker 中的网络不仅是网络，还是 Docker 所特有的一个子模块。Docker 中的网络可以联通容器，也能让容器访问到外部网络，以及被外界访问。

### 12.2.1 默认网络

通过 `docker network ls` 命令，可以获得当前 Docker 中所有网络的列表。安装和首次启动 Docker 时，Docker 会自动创建三个默认网络，在这种情况下，通过 `docker network ls` 命令可以看到这三个默认网络。



```
$ sudo docker network ls
```

NETWORK ID	NAME	DRIVER	SCOPE
3db7a5fb9797	bridge	bridge	local
56c9379a0b56	host	host	local
8657fd90fc0e	none	null	local

这三个网络都是 Docker 的默认实现，bridge 网络是 Docker 容器中默认使用的网络。在宿主机的网络中，它对应着 docker0 网络，一旦安装了 Docker，就可以看到 Docker 在宿主机中创建的 docker0 网络。docker0 在 Docker 中扮演着网桥的角色，也就是容器网络模型中的网络。

```
$ sudo ifconfig
docker0: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
    inet 192.168.0.1 netmask 255.255.240.0 broadcast 0.0.0.0
    ether 02:42:2b:50:17:ce txqueuelen 0 (Ethernet)
    RX packets 8867 bytes 513596 (501.5 KiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 15185 bytes 22695874 (21.6 MiB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

在默认情况下，创建的容器会被 Docker 连接到 bridge 这个网络上，这个网络所使用的正是宿主机中的 docker0 网。如果想改变容器使用的网络，可以在创建容器时使用 --network 参数进行修改。

```
$ sudo docker run -it --name nonenetwork --network none ubuntu /bin/bash
```

none 网络表示不使用网络。容器如果绑定在 none 网络上，则 Docker 不会为容器分配带有网络连接的网络栈，而是通过一个无连接的网络让容器处于与外界网络环境完全隔离的状态。

host 网络直接使用宿主机的网络环境。容器如果绑定在 host 网络中，则 Docker 会直接采用宿主机内的网卡作为容器的网络连接对象。也就是说，其他与宿主机同在一个子网的机器也能发现容器的存在。

通过 docker network inspect 命令，可以了解容器网络比较详细的信息。

```
$ sudo docker network inspect bridge
[
  {
    "Name": "bridge",
    "Id": "3db7a5fb97979aec6028168897635ea15elc9be63cfa8e650e6b37ef7d31e0b4",
    "Scope": "local",
    "Driver": "bridge",
```

```

"EnableIPv6": false,
"IPAM": {
  "Driver": "default",
  "Options": null,
  "Config": [
    {
      "Subnet": "172.17.0.0/16",
      "Gateway": "172.17.0.1"
    }
  ]
},
"Internal": false,
"Containers": {
  "3e9df8c1b6250d5619db98a9862fe83d2f4091686d876b5deef934b100bfb04": {
    "Name": "container2",
    "EndpointID": "5f6af8bf453a58f89f09ebafebd7a5a8122c1f97ecb6269ecbbacdbf910e326a",
    "MacAddress": "02:42:ac:11:00:03",
    "IPv4Address": "172.17.0.3/16",
    "IPv6Address": ""
  },
  "8d64a2fdcf84336e3d0900fe14311056c4580fa666a77b1c9e39b0809f12383e": {
    "Name": "container1",
    "EndpointID": "b47b5163241021212c8bd4c78802706a46cb01915092b4b22c0b3285b63d05d3",
    "MacAddress": "02:42:ac:11:00:02",
    "IPv4Address": "172.17.0.2/16",
    "IPv6Address": ""
  }
},
"Options": {
  "com.docker.network.bridge.default_bridge": "true",
  "com.docker.network.bridge.enable_icc": "true",
  "com.docker.network.bridge.enable_ip_masquerade": "true",
  "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
  "com.docker.network.bridge.name": "docker0",
  "com.docker.network.driver.mtu": "1500"
},
"Labels": {}
}
]

```

在网络信息中展示的 `Containers` 字段，包含的就是连接到这个网络的所有容器，包括每个容器所分配的端点 ID、物理地址、IP 地址等。我们看到，由于同在一个网络内的容器存在于同一个网段中，所以它们就可以互相进行访问。

## 12.2.2 自定义网络

如果没有为容器选定网络，则 `Docker` 会将新创建的容器连接到 `bridge` 默认网络上。这样在一个网络中的容器，就有了互相进行访问的可能性，这在某些情况下并不是我们所期望的。有时，对于由数个容器所组成的一个小型模块，我们希望这些容器只能相互访问，而不能访问其他容器，也不能被其他容器访问。此时，就需要为这几个容器单独分配网络，让它们连接到一个独立的网络中，从而隔绝其他网络中的容器对这几个容器的连接。

在将容器放入单独的网络之前，要先创建一个网络供容器进行连接。通过 `docker network create` 命令可以创建网络。

```
$ sudo docker network create --driver bridge isolated
782a57f947530ebc173159fd723665d9e67e88a16c6e55980b07a0c3b662c3
```

`--driver` 参数用来指定网络所基于的网络驱动，也可以将其简写为 `-d`。在默认情况下，`Docker` 会采用 `bridge` 网络作为网络驱动。容器网络模型的建立，使我们可以根据自己的需要寻找更多符合需求的网络驱动。只要按照容器网络模型进行设计，任何人都能开发出适用于 `Docker` 的网络驱动。

网络模型创建完成之后，可以在 `Docker` 的网络列表中找到创建好的网络。

```
$ sudo docker network ls
```

NETWORK ID	NAME	DRIVER	SCOPE
3db7a5fb9797	bridge	bridge	local
56c9379a0b56	host	host	local
782a57f94753	isolated	bridge	local
8657fd90fc0e	none	null	local

使用 `docker network inspect` 命令可以得到网络更详细的信息。

```
$ sudo docker network inspect isolated
[
  {
    "Name": "isolated",
    "Id": "782a57f947530ebc173159fd723665d9e67e88a16c6e55980b07a0c3b662c3",
    "Scope": "local",
```



```
"Driver": "bridge",
"EnableIPv6": false,
"IPAM": {
  "Driver": "default",
  "Options": {},
  "Config": [
    {
      "Subnet": "172.18.0.0/16",
      "Gateway": "172.18.0.1/16"
    }
  ]
},
"Internal": false,
"Containers": {},
"Options": {},
"Labels": {}
}
```

我们可以从网络的子网段及网关等信息中发现，新创建的网络已经与之前看到的 bridge 默认网络不在同一个网络之中了，所以连接到 isolated 网络中的容器与连接到 bridge 网络上的容器，无法直接借助所在的网络进行通信。

### 12.2.3 容器与外部通信

在默认情况下，容器位于 Docker 管理的 docker0 网桥中，这个网桥搭建在宿主机的虚拟网卡上，不与宿主机的其他网络挂钩。也就是说，容器所在的网络环境其实是隔绝在宿主机内的，宿主机外的计算机和程序是无法发觉和发现容器网络的存在，也无法连接到容器上。

数据从容器网络传输到 docker0 或其他宿主机上的虚拟网卡后，会被转发到其他的宿主机网卡上。如果这些网卡能够访问外部网络，那么容器自然也就拥有了访问外部网络的权利。

要使容器与外部通信都正常运作，最关键的就是要保证网络数据转发，也就是 IP forward 功能正常启用。Docker daemon 启动时，我们可以通过--ip-forward 参数来控制 Docker 是否使用 IP forward，默认配置是开启使用，所以通常情况下不需要对其专门进行设置。如果 Docker daemon 已经开启了对 IP forward 的支持，但容器仍然无法连接外部网络，可以检查宿主机系统中的 IP forward 是否被禁用。

```
$ sudo sysctl net.ipv4.conf.all.forwarding
net.ipv4.conf.all.forwarding = 0
```

当这一参数为 0 时,表示系统内核的 IP forward 处于禁用状态,开启它才能使 Docker 进行 IP forward 操作。

```
$ sudo sysctl net.ipv4.conf.all.forwarding=1
$ sudo sysctl net.ipv4.conf.all.forwarding
net.ipv4.conf.all.forwarding = 1
```

容器可以通过虚拟网卡在宿主机内的转接,访问到宿主机外部的网络。外部网络想要访问容器,则需要实现端口映射。只有将容器的端口映射到宿主机上,位于宿主机外部网络上的计算机和其中的程序,才能够通过访问宿主机的方式访问到容器中。

在之前的章节里提到的实现外部与容器通信的端口映射方案,是基于 Iptables 这个防火墙的,更确切地说,是基于 Iptables 中的 DNAT (Destination Network Address Translation, 目标地址转换) 的。

当我们通过启动容器时传递的 -P 或 -p 参数使容器内的端口映射到宿主机上时, Docker 会在 Iptables 中增加一条通过容器网络连接到容器上的 DNAT。在 Iptables 的规则中,可以看到这条转发记录。

```
$ sudo iptables -t nat -L -n
...
Chain DOCKER (2 references)
target                prot opt source                destination
DNAT                  tcp  --  0.0.0.0/0              0.0.0.0/0              tcp dpt:443 to:192.168.32.6:443
DNAT                  tcp  --  0.0.0.0/0              0.0.0.0/0              tcp dpt:80 to:192.168.32.6:80
...
```

这里映射了容器的 80 和 443 端口,在 Iptables 的规则列表中可以发现相应的转发规则。转发源是在 -p 参数没有传入具体的监听地址时默认的全部地址 0.0.0.0,目标地址是容器在容器网络中的地址和端口。在本例中, Docker 为容器在网络中所分配的地址为 192.168.32.6。

使用 -p 参数能让我们选定宿主机上指定的端口进行绑定,而使用 -P 参数则会使 Docker 从宿主机上寻找可以绑定的端口进行绑定。那么 Docker 是如何从宿主机中找到这个可以被映射的端口的呢?宿主机系统可以被分配的端口会出现在 /proc/sys/net/ipv4/ip\_local\_port\_range 这个文件中, Docker 正是从中找出端口进行映射的。

```
$ sudo more /proc/sys/net/ipv4/ip_local_port_range
32768    61000
```

在这个用于记录可以被绑定端口的系统文件里，标记了一个能够被绑定端口的范围，这个范围通常就是 32768~61000。

## 12.2.4 容器间通信

容器间通信的配置，我们在之前的章节中已经进行了阐述，即通过--link 参数指定需要进行访问的容器。在容器间可进行连接的配置建立之后，可以在容器的信息中发现相关的条目。

```
$ sudo docker create --name php php
fa400be237f1fb5463e0d42e1803c73f09c14a33a67c30085835394267862616
$ sudo docker create --name nginx --link php nginx
e795f4643ael8a3cdeb4ea62a5a4382db7c74a45acdbae6a6f8d492f7574951d
$ sudo docker inspect -f "{{ .HostConfig.Links }}" nginx
[/php:/nginx/php]
```

在容器的信息中，展示了连接到的容器，以及这个容器在当前容器中的别名。

容器间通信的主要目的并不是实现网络的访问，而是将容器间访问的方式更抽象化。我们知道，Docker 设计的目的是快速部署，而快速部署需要解决的问题就是减少和避免因为宿主机环境变化而带来的配置和适配时间问题。网络作为宿主机环境的重要组成部分之一，自然也是影响 Docker 是否能够完成快速部署最重要的因素之一。

同在一个容器网络中的容器因为连接了同一个子网，所以它们之间进行网络访问是非常容易的。然而由于宿主机的网络环境并不固定，也就不能保证 Docker 申请到的容器网络的网段总是一致。如果让容器中的程序以其他容器在网络中的 IP 地址进行访问，则势必会因宿主机网络环境的改变而引起容器网络的改变，进而导致网络访问并不牢靠。在这种情况下，就需要修改容器中访问其他容器所使用的 IP 地址，但这样做就达不到 Docker 快速部署的目的。

Docker 通过修改 hosts 的方式实现了一种既简单又无须修改 IP 地址的方案。我们使用 Docker 连接其他容器时，Docker 会在/etc/hosts 中添加一条基于容器名称或别名的条目，这条解析的指向正是被连接的容器。当我们需要在容器中使用被连接容器地址的时候，只使用容器的名称或设置的别名即可。这种方式巧妙地利用了域名解析机制实现了变化的 IP 到固定的名称的转变，无须再考虑容器 IP 因为网络环境而发生变化的问题。

当我们创建容器并且使用了容器间连接时，Docker 会在容器中做两件事情。第一件事情是修改/etc/hosts 文件，第二件事情是增加相关的环境变量。



我们可以在容器中查看对/etc/hosts 文件进行的修改。

```
root@e795f4643ae1:/# cat /etc/hosts
127.0.0.1      localhost
...
192.168.0.2    php fa400be237f1
...
```

我们可以看到，被连接容器的容器 ID 前 12 位及容器名称都会被解析到被连接容器的 IP 上。

在容器的环境变量中，我们也能更具体地看到被连接容器的信息。

```
root@e795f4643ae1:/# env
...
PHP_PORT=tcp://192.168.0.2:9000
PHP_PORT_9000_TCP_PROTO=tcp
PHP_PORT_9000_TCP_PORT=9000
PHP_PORT_9000_TCP=tcp://192.168.0.2:9000
PHP_NAME=/nginx/php
PHP_PORT_9000_TCP_ADDR=192.168.0.2
...
```

在环境变量中不但可以找到被连接容器的连接名称、实际 IP 地址的信息，还能找到每个被连接容器所暴露端口的端口号、网络协议及连接使用的字符串等。

## 12.3 网络实践

Docker 为管理网络提供了丰富的方法，通过管理和控制网络命令，能够轻松地将网络连接容器上，让网络为容器提供服务，也能对 Docker 网络进行深层次的定制。

### 12.3.1 管理容器网络

之前我们简单谈到过几个用来操作 Docker 网络的命令，这里我们进行进一步的探讨。Docker CLI 为我们提供了 4 个用于管理容器网络的命令，分别用于创建网络、获得网络列表、获得网络信息、删除网络。

在之前的小节中我们已经展示了通过 `docker network create` 来创建网络，在创建网络

的同时，我们还能通过一些参数来控制所创建网络的配置。下面使用 `--subnet` 参数创建一个拥有指定子网范围的容器网络。

```
$ sudo docker network create --subnet 10.10.200.0/24 cnet
c22ee8e4b38ce34a82c3d5bd5f7d4b7029cddf5bdb8379458daf1bd36b26bb18
```

注意，在不使用 `-d` 参数指定网络所使用的网络驱动时，Docker 会默认使用 `bridge` 驱动，也就是通过 `docker0` 网桥实现容器网络。

```
$ sudo docker network ls
```

NETWORK ID	NAME	DRIVER	SCOPE
a3b616dda81b	bridge	bridge	local
c22ee8e4b38c	cnet	bridge	local
56c9379a0b56	host	host	local
8657fd90fc0e	none	null	local

在 `docker network ls` 命令展示出的 Docker 中的容器网络列表中，已经能够看到刚刚创建的容器网络了，通过 `docker network inspect` 命令，可以得到这个容器网络的详细信息。

```
$ sudo docker network inspect cnet
[
  {
    "Name": "cnet",
    "Id": "c22ee8e4b38ce34a82c3d5bd5f7d4b7029cddf5bdb8379458daf1bd36b26bb18",
    "Scope": "local",
    "Driver": "bridge",
    "EnableIPv6": false,
    "IPAM": {
      "Driver": "default",
      "Options": {},
      "Config": [
        {
          "Subnet": "10.10.200.0/24"
        }
      ]
    },
    "Internal": false,
    "Containers": {},
    "Options": {},
    "Labels": {}
  }
]
```

在 `docker network inspect` 命令展示出的容器网络信息中，可以看到网络的子网范围正是我们所设置的值。

在容器网络未被使用的情况下，可以通过 `docker network rm` 命令对容器网络进行移除。

```
$ sudo docker network rm cnet
cnet
```

## 12.3.2 容器连接网络

要让容器使用指定的容器网络，可以在创建容器时使用 `--network` 参数进行设置。

```
$ sudo docker run -it --network cnet busybox
```

使用 `--network` 参数时，可以指定 Docker 提供的三个默认网络：bridge、host、none，也能使用在 Docker 容器网络列表中所存在的网络名称。另外，还可以通过 `container:<name|id>` 这种形式，传入容器的名称或容器的 ID，这样新的容器就会使用与这个容器相同的网络。

除了在创建容器时为容器选择网络，还能够随时通过 `docker network connect` 命令让容器连接到指定的网络上。这里我们将创建的 `cnet` 网络连接到一个名为 `busybox` 的容器上。

```
$ sudo docker network connect cnet busybox
```

连接之后查看 `cnet` 网络的信息。

```
$ sudo docker network inspect cnet
[
  {
    "Name": "cnet",
    "Id": "8blb85fdf4538803c2796edlefa41fbff8acfb74c4403d7809c62f2c44fa901",
    "Scope": "local",
    "Driver": "bridge",
    "EnableIPv6": false,
    "IPAM": {
      "Driver": "default",
      "Options": {},
      "Config": [
        {
          "Subnet": "10.10.200.0/24"
```



```

    }
  ],
  "Internal": false,
  "Containers": {
    "427995ede694f93b542e364f4ede1055b43c17f0df77b02cc0587d81f9b7ce50": {
      "Name": "busybox",
      "EndpointID": "2f2ebed3e3f7650ca27262858a8b768ce76a2868931322eec3516de59c06c8e4",
      "MacAddress": "02:42:0a:0a:c8:02",
      "IPv4Address": "10.10.200.2/24",
      "IPv6Address": ""
    }
  },
  "Options": {},
  "Labels": {}
}
]

```

我们可以看到，网络的信息中已经显示出了连接到此网络的 busybox 容器。我们进入到容器中检查容器内的网络环境。

```

$ sudo docker exec -it busybox sh
/ # ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:AC:11:00:02
          inet addr:172.17.0.2  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe11:2/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:8 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:648 (648.0 B)  TX bytes:648 (648.0 B)

eth1      Link encap:Ethernet  HWaddr 02:42:0A:0A:C8:02
          inet addr:10.10.200.2  Bcast:0.0.0.0  Mask:255.255.255.0
          inet6 addr: fe80::42:aff:fe0a:c802/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:8 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:648 (648.0 B)  TX bytes:648 (648.0 B)

lo        Link encap:Local Loopback

```

```

inet addr:127.0.0.1 Mask:255.0.0.0
inet6 addr: ::1/128 Scope:Host
UP LOOPBACK RUNNING MTU:65536 Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1
RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

```

在容器内的网络环境中，除了一个本地网络，还存在两个以太网，分别使用了 172.17.0.0 和 10.10.200.0 两个子网。而这两个网络，分别对应着创建容器时 Docker 自动分配的 bridge 网络，以及连接到容器的 cnet 网络。

同时，也能够通过 `docker network disconnect` 命令随时将容器网络与容器断开。可以说，通过 Docker 进行容器与网络之间连接的操作，就如同插拔网线一样方便。

```
$ docker network disconnect cnet busybox
```

### 12.3.3 配置 docker0 网桥

使用 Docker 时，Docker 会在宿主机中建立一个默认的网桥网络，也就是 `docker0` 网络。通过 `docker0` 网络接口，可以向宿主机的其他网络接口发送数据包，并得到它们所返回的结果。这一桥接可以发生在任何宿主机的网络接口上，不论它们是物理的还是虚拟的。`docker0` 网桥的桥接，让容器能够通过宿主机的对外网络访问到互联网中的内容。

Docker 管理着 `docker0` 网络的配置，如 IP 地址、子网掩码、网段等基本信息。虽然我们能够在宿主机中修改 `docker0` 网络的配置，但是由于这些配置直接接受 Docker 的管理，并且修改它会影响 Docker 的运行，所以不建议通过 Docker 以外的方式去修改 `docker0` 的配置。

由于 Docker 服务一旦启动，就表示随时会有容器运行在 Docker 之中，为了避免容器运行时修改网络带来的不稳定性，配置 `docker0` 的工作只能在 Docker 服务启动时进行。

通过 `docker daemon` 或 `dockerd` 命令启动 Docker daemon 程序时，可以指定与 Docker 管理的 `docker0` 网络相关的配置。例如，可以通过 `--bip` 参数设置 `docker0` 自身所使用的网络 IP。

```
$ sudo dockerd --bip=192.168.1.1/24
```

通过 `--fixed-cidr` 参数，能够配置 `docker0` 的子网网段。

```
$ sudo dockerd --fixed-cidr=192.168.1.0/24
```

除了设置基础网络的基本信息,还可以进行网络限流等方面的设置。例如,使用`--mtu`参数可以设置 `docker0` 的最大数据包长度。

```
$ sudo dockerd --mtu=65536
```

在宿主机中运行 `brctl show` 命令,能够显示宿主机中网桥网络建立的连接。而 `docker0` 作为一个存在于宿主机中的网桥网络,自然也能够被这个命令所展示。

```
$ sudo brctl show
bridge name bridge id            STP enabled  interfaces
docker0      8000.0242a1d49bd1 no           veth1cb50ac
              veth9342c99
              vethc35da16
```

在 `brctl show` 命令所显示的信息中,可以看到 `docker0` 上连接的三个接口,它们都为 Veth 通道,这说明它们正连接着三个运行在 Docker 中的容器。

每当我们创建容器而没有指定容器网络时,也就是为容器选取 bridge 默认网络时,Docker 会从 `docker0` 中找到一个未被分配的 IP 地址,将其连接到容器上。同时,Docker 也会将容器中虚拟的 `eth0` 网络接口中的 IP 地址等信息,设置为与 `docker0` 相对应的信息。这样,虽然容器是通过 Veth Pair 这类通道进行网络数据交换的,但是在程序感知上,容器是直接连接到 `docker0` 上的。

所以,当程序运行在容器中时,也可以通过 `eth0` 网络接口信息来了解容器在 `docker0` 中分配的网络配置。

```
$ docker run -it --rm ubuntu /bin/bash
root@d5ca0dc35a2a:/# ip addr show eth0

24: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen
1000
    link/ether 32:6f:e0:35:57:91 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.3/16 scope global eth0
        valid_lft forever preferred_lft forever
    inet6 fe80::306f:e0ff:fe35:5791/64 scope link
        valid_lft forever preferred_lft forever

root@d5ca0dc35a2a:/# ip route

default via 172.17.42.1 dev eth0
172.17.0.0/16 dev eth0 proto kernel scope link src 172.17.0.3
```



### 12.3.4 自定义网桥

虽然能通过 Docker 进行 `docker0` 网桥的设置, 但能够设置的范围有限, 有时是不能满足我们的需求的。在这种情况下, 我们可以在宿主机中自定义一个新的网桥, 以替换 `docker0` 网桥的方式来实现更多对网桥的配置。

通过 `brctl` 和 `ip` 命令, 可以完成在宿主机中创建网桥和配置网桥的工作。

```
$ sudo brctl addbr ymbr0
$ sudo ip addr add 192.168.99.1/24 dev ymbr0
$ sudo ip link set dev ymbr0 up
```

可以通过 `ip addr show` 命令显示刚刚创建的网桥接口的相关信息。

```
$ sudo ip addr show ymbr0
```

在 Docker 没有启动时, 可以通过启动 Docker `daemon` 命令并携带 `-b` 或 `--bridge` 参数的方式使 Docker 采用指定的网桥。如果 Docker 已经启动了, 则需要先将 Docker 停止, 再替换原有的 `docker0` 网桥。

```
$ sudo dockerd --bridge ymbr0
```

通过替换默认的 `docker0` 网桥, 我们能够完成更多、更具体的需求。这里我们只进行替换网桥的介绍, 对于如何配置宿主机中的网桥, 以及通过这些自定义又能完成怎样的需求, 已经超出了本书的研究范围, 大家可以从互联网或者其他书籍中寻找关于这些问题的具体答案。

### 12.3.5 配置 DNS

DNS 是域名解析服务, 是网络访问中最常用的一种服务, 通过 DNS 可以省去要记住没有规律的 IP 地址的麻烦, 也能防止因为 IP 地址更换而带来的耦合效应。通常, 在系统中使用 DNS 功能的方式主要有两种, 一种是通过 `hosts` 文件在本地进行域名解析, 一种是通过 DNS 服务器从互联网中解析域名。

在 Linux 系统中, 与 DNS 解析相关的配置文件主要有三个, 分别是 `etc` 目录下的 `hostname`、`hosts` 和 `resolv.conf`。`hostname` 文件用于给出主机名, 主要是在其他主机进行网络发现时告知对方自身的名称。`hosts` 文件用来表示本地域名解析时的解析列表, 它的内容就是域名及对应的解析 IP。`resolv.conf` 文件的作用是提供 DNS 服务器的列表, 用于在本地解析无效时提供向互联网请求解析时所要连接的服务器地址, 并且对这些地址的先后顺序进行处理。

Docker 容器的文件系统直接基于对应的基础镜像所建立，这三个文件自然也会存在于 Docker 镜像之中。我们知道，这三个文件中的参数会因运行环境的不同发生改变：`hostname` 文件中要写入容器对应的主机名，`hosts` 文件中有时也会附加容器间连接的参数，`resolv.conf` 文件的配置要参考网关给出的值。因此，直接通过镜像继承这三个文件是达不到期望的效果的。

在 Docker 里对这三个与 DNS 有关的文件进行了特殊处理，我们可以在容器里运行 `mount` 命令看出其中的端倪。

```
root@4199e5a43092:/# mount
...
/dev/sda2 on /etc/resolv.conf type ext4 (rw,relatime,data=ordered)
/dev/sda2 on /etc/hostname type ext4 (rw,relatime,data=ordered)
/dev/sda2 on /etc/hosts type ext4 (rw,relatime,data=ordered)
...
```

可以看到，这三个文件都以挂载的形式存在。也就是说，Docker 通过挂载覆盖了在容器文件系统上的对应文件，以让 Docker 能够更方便地对这三个文件及其中的内容进行管理。

以这种方式进行设计，最直接的受益者应该是 `resolv.conf` 文件。在通常情况下，主机是以 DHCP 的形式连接到网络中的，DNS 服务器也是由主机自动获取 DHCP 所给出的 DNS 服务器设置的。若网络发生改变，主机会重新获取新的 DNS 服务器，以保障 DNS 解析能够继续正常工作。此时，虽然宿主机所连接的网络进行了更新，但 Docker 容器所使用的 `docker0` 网桥却没有任何更改，所以运行在宿主机中的 Docker 容器就不会进行 DNS 服务器的自动更新了。由于有了 `resolv.conf` 的挂载，Docker 能够在宿主机网络环境改变的时候，很快地将相应的配置应用到容器之中，实现了容器实时更新 DNS 服务器的目的。

同时，在创建容器时指定相关参数，也能够对与容器内 DNS 解析相关的内容进行配置。

通过 `-h` 或 `--hostname` 参数可以配置容器的主机名。这一配置会写入 `/etc/hostname` 中，使用 `/bin/bash` 程序时，也能够看到主机名的显示。在默认情况下，Docker 会采用容器 ID 的前一部分作为容器的主机名，而通过刚才提到的 `-h` 或 `--hostname` 参数也可以修改这个默认的配置。

```
$ sudo docker run -it --rm ubuntu cat /etc/hostname
2c83a63b17bd
$ sudo docker run -it --rm -h ymdot ubuntu cat /etc/hostname
ymdot
```

通过 `--link` 参数可以连接其他容器，建立容器间网络通信。使用这个配置会在 `/etc/hosts` 文件中生成基于给定的连接容器名称或别名的字段，而其指向的是被连接容器的实际 IP 地址。

通过 `--dns` 参数能够为容器指定新的 DNS 服务器，这个配置会写入 `/etc/resolv.conf` 中，我们可以给出多个 `--dns` 参数，每一个参数在 `/etc/resolv.conf` 文件中占一行。

通过 `--dns-search` 参数可以指定 DNS 的搜索域，这个配置也会写入 `/etc/resolv.conf` 文件中。DNS 搜索域是指对给出的域名，除了在主域中搜索其地址，还在搜索域下进行搜索。例如我们将搜索域定为 `ymdot.cn`，那么容器在解析 `www` 时，不但会从 DNS 服务器中寻找 `www` 这个域名对应的 IP 地址，也会寻找 `www.ymdot.cn` 对应的 IP 地址。

### 12.3.6 使用 IPv6

在默认情况下，Docker 会为容器创建基于 IPv4 的网络，并且只会创建基于 IPv4 的网络。虽然 Docker 对 IPv6 也进行了支持，但我们需要通过配置来启动它的支持。

我们可以通过在启动 Docker daemon 时附加相关的参数，来开启 Docker 对 IPv6 的支持。用于开启 Docker 对 IPv6 支持的参数为 `--ipv6`，只要在 `dockerd` 命令中传入这个参数就能使运行起来的 Docker daemon 使用到 IPv6 的网络通信了。

```
$ sudo dockerd --ipv6
```

由于 Docker 容器默认会连接到 `docker0` 网桥上，所以要保证容器正常使用 IPv6 进行通信，让 `docker0` 网桥支持 IPv6 是必不可少的。由于 `docker0` 网桥接受 Docker 的直接管理，所以在开启 Docker daemon 对 IPv6 支持的同时，Docker 也会使 `docker0` 提供对 IPv6 的支持。开启 IPv6 时，Docker 会为 `docker0` 网桥分配一个默认的 IPv6 地址，这个地址通常为 `fe80::1`，可以通过 Docker daemon 的启动参数 `--fixed-cidr-v6` 修改这一默认值。

```
$ sudo dockerd --ipv6 --fixed-cidr-v6="2001:db8:1::/64"
```

与 Docker 中的 IPv4 协议一样，Docker 会让容器对外访问的数据通过 IP 转发的方式，从 `docker0` 网桥传递到宿主机其他的网络接口上。我们可以通过设置 Docker daemon 启动参数中的 `--ip-forward` 为 `false` 关闭数据转发，使容器只能对 `docker0` 所搭建的子网内的终端进行访问，而不能访问宿主主机上其他网络中的机器，也无法通过宿主机对外的网络接口访问外网乃至互联网中的信息。

开启 Docker 对 IPv6 的支持之后，每一个新创建的容器都会得到从 `docker0` 子网中所



分配的一个 IPv6 的 IP 地址,我们能够从容器内的网络接口信息中找到容器被分配的 IPv6 配置的信息。

```
$ sudo docker run -it ubuntu /bin/bash -c "ip -6 addr show dev eth0; ip -6 route show"

15: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500
    inet6 2001:db8:1:0:0:242:ac11:3/64 scope global
        valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe11:3/64 scope link
        valid_lft forever preferred_lft forever

2001:db8:1::/64 dev eth0 proto kernel metric 256
fe80::/64 dev eth0 proto kernel metric 256
default via fe80::1 dev eth0 metric 1024
```

## 12.4 本章小结

在互联网时代,网络几乎是所有软件程序不可或缺的支持,也有相当一部分程序本身就是针对和依赖网络开发的。同时,借助网络将大型服务拆解成小服务并分散到不同的机器上,也是所有企业逐步推进的方向。对于能够进行快速部署的 Docker 来说,提供一个稳定性高、适应性强、使用简单的网络架构是它称霸容器虚拟化领域的尚方宝剑。

在本章中,我们更加深入地了解了 Docker 的网络模块,对容器网络模型,以及 Docker 对它的具体实现方式进行了解读。掌握了 Docker 网络模块的基本知识之后,我们还对几个 Docker 网络部分的配置方法进行了说明。虽然篇幅有限,不能把所有关于 Docker 网络控制的方法进行逐一讲解,但是在熟悉了 Docker 网络的基础知识和实现方式后,大家可以举一反三,在进行与 Docker 网络相关的配置等场景里,得心应手地使用 Docker 网络模块。

# 第 13 章

## 安全加固

安全是一个老生常谈的问题，但却实实在在地关系到程序运作的稳定、数据的保护等许多方面。Docker 是在 Linux Container 技术的基础上实现的，与其他大多数虚拟化方案有所不同，运行在容器中的进程也是真实运行在宿主机中的。容器中的进程只是通过 Namespaces 进行简单的运行信息隔离，但它与宿主机中的进程其实没有太大的区别。也就是说，如果没有使用合适的安全策略，就很有可能出现恶意程序跳出容器隔离的环境，影响宿主机系统环境的情况。

从安全的角度来说，容器进程只是很小的一方面，其在网络、文件权限、能力控制及资源调用等方面，都是非常重要的。Docker 的安全性在很大程度上取决于 Linux 系统的安全策略。所以，我们操作和加固 Docker 安全的方式，主要通过 Linux 系统本身的安全机制及工具软件来实现。

Docker 的安全性是在实际生产中衡量整个服务体系，更是基础架构最重要的因素之一。所以，做好 Docker 的安全防护也是搭建和使用 Docker 需要掌握的必要知识。在本章中，我们就专门对 Docker 的安全防护策略进行简要介绍。

### 13.1 深入理解 Docker 安全

安全是程序运行不可或缺的部分，Docker 在安全性上也提供了丰富的支持。Docker

通过多位一体的方式将众多安全防护策略和安全防护软件进行组合,为 Docker 容器提供安全保护。

### 13.1.1 命名空间隔离

如图 13-1 所示, Docker 所基于的 Linux Container 是通过 Linux 内核提供的 Namespaces 技术,运行在不同容器中的程序进行隔离的。和虚拟机实现相比,Namespaces 提供的隔离其实是非常基础的,仅仅使用了简单的方式,就让不同命名空间独立享有进程信息、网络信息等。

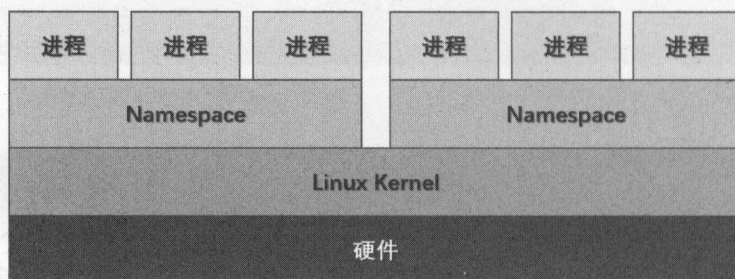


图 13-1 命名空间隔离

相较于 Hypervisor 实现的隔离,Namespaces 提供的隔离其实是非常基础的。也正是因为 Namespaces 只提供了简单直接的隔离,运行在 Namespaces 中的程序几乎没有任何的指令转换等操作,因此它能够带给我们更强的运行效率。但事情总不是十全十美的, Linux Kernel 自 2008 年 7 月引入 Namespaces 开始,虽然经历了多年的打磨和改进,也将其应用到了许多实际生产中,但它仍存在一定的风险因素。

被隔离在 Namespaces 中的程序虽然无法感知命名空间以外的程序存在,但由于 Namespaces 结构设计的特性,运行在命名空间(Docker 容器)中的程序,仍可以直接访问宿主机系统内核的功能和部分核心文件。

特别是网络部分,它可以说是内核功能中最常被调用的模块。为了让容器使用网络通信, Docker 使用 Veth Pair 打穿了 Network Namespaces。这就意味着,即使命名空间机制隔离了容器之间的网络,让每个容器都拥有了自己的网卡、网络栈、网络配置等,但这并不意味着它们就不能直接访问宿主机的网络了。因为从网络架构上来说,容器与外界进行网络连接所使用的网桥,其实属于宿主机网络的一部分。

在虚拟机实现的隔离中,应用程序的资料调用都通过 Hypervisor 层来实现,所以只



要在 Hypervisor 层加上一定的权限控制机制,就能阻止不安全的程序直接进行系统调用。而在 Namespaces 的实现中,直接做中间拦截并不方便。所以,需要通过专门的手段,限制容器中程序对系统内核的调用。

### 13.1.2 资源控制组

控制组 (CGroup) 与 Namespaces 一样,是组成容器技术的核心模块之一。控制组的主要功能就是对 CPU、内存、硬盘 IO 等程序访问的计算机资源进行控制。在 Docker 的实现中,每个容器都拥有一个独立的控制组策略集,用来控制容器中每个程序对计算机资源的访问和调用。通过控制组,我们可以实现以下功能。

- ❑ 资源限制:对计算机资源进行分配,控制程序对资源的占有量。
- ❑ 优先化:实现程序之间资源分配的差异化,让部分程序优先得到资源。
- ❑ 用量报告:准确得到每个控制组中分配资源的统计信息。

命名空间机制限制了程序之间的访问,避免了恶意程序干扰其他容器中程序的运行。而控制组则限制了程序对计算机资源的使用,能够避免问题应用挤占计算机资源,导致同一宿主机的其他容器中的程序,由于资源分配不足出现的问题。

除了防止利用漏洞入侵被上传的恶意程序,以及由于程序设计不仔细造成问题程序对资源的不正当消耗,在防御一些外部攻击上,控制组也有不错的效果。例如,在防御 DDoS 攻击中,控制组就能很好地保证部分服务出现瘫痪时,其他的不会受到牵连和影响。对于大型云平台或分布式集群,控制组所扮演的角色其实是十分重要的。

### 13.1.3 内核能力机制

命名空间不能很好地阻止容器内程序对系统核心的调用,所以需要另外的机制保障宿主机系统的安全,内核能力机制就是其中之一。能力机制是 Linux 内核中进行细粒度权限管理的模块,相较于传统的权限控制,能力机制能够更好地管理不同程序,甚至调用不同内核功能所对应的权限。能力机制主要完成了以下两项工作。

- ❑ 权限检查:内核必须检查进行所有特权操作的进程是否拥有对应的权限,阻止没有权限的程序对系统核心部分进行的修改。
- ❑ 复原更改:对核心文件与配置提供恢复修改的支持。

Docker 已经通过内核机制限制了容器中的程序使用内核,在默认情况下,Docker

只允许程序使用 `kill`、`setgid`、`setuid`、`net_bind_service`、`net_raw`、`chown` 等部分内核能力。

通过内核能力的帮助，加强了 Docker 对系统操作的安全性。例如，如果程序希望绑定端口，只需要分配给它 `net_bind_service` 的能力，不需要赋予它完全的根权限。这就很好地保障了攻击者无法在容器中取得根权限，大幅降低了对宿主机造成危害的可能性。

当然，内核能力机制虽好，但也要用得对才能展现出它的优势，即要对症下药。在早期的 Docker 中，就出现过因为分配内核能力不恰当，造成了容器内的程序可以通过系统调用，访问到宿主机系统内部任意文件的情况。

## 13.2 资源使用限制

限制资源使用是避免因程序异常而导致宿主机引起连锁反应的重要手段，也是避免容器之间相互干扰、抢占资源的主要方式。

### 13.2.1 通过控制组限制

在 Docker 中，实现了能够通过控制组限制容器对计算机资源调用的作用。

#### 1. 内存

内存是程序中不可以获取的计算资源，为了避免单一容器因为故障用尽宿主机的所有内存，导致其他容器中运行的程序因没有内存可用而出现连锁反应，可以限制单一容器能分配的最大内存。在创建容器时带入 `-m` 或 `--memory` 参数，并加上表示内存容量的数字，就能限制容器的内存用量。

```
$ sudo docker run -d -m 512M nginx:stable
```

在 Linux 中，内存分为物理内存和交换区内存，当物理内存不足时，会将部分内存中的数据移动到交换区中，以腾出内存供程序使用。使用 `-m` 或 `--memory` 参数，其实只限制了物理内存的使用，并没有限制交换区内存的使用。所以，问题程序仍然能够占满交换区内存。

通过 `--memory-swap` 参数，可以设置容器对总内存的使用，也就是物理内存加上交换区内存的总量。将 `-m` 或 `--memory` 参数组合使用，就可以达到控制容器使用物理内存和交换区内存大小的目的了。

```
$ sudo docker run -d -m 512M --memory-swap 1024M nginx:stable
```

除了限制内存，限制 CPU 也是避免问题程序影响整个宿主机稳定的方法之一。由于不同型号的 CPU 参数差异较大，所以在 Docker 里，没有直接使用具体的参数进行配置，而是通过权重来划分 CPU 分配。我们在创建容器时，使用 `-c` 或 `--cpu-shares` 参数就能设置容器占用的 CPU 权重了。

```
$ sudo docker run -d -c 500 nginx:stable
$ sudo docker run -d -c 1000 php:7-fpm
```

此处我们把 Nginx 和 PHP 服务所占用的 CPU 资源占用比例配置为 1:2，也就是说，这两个容器可以得到的 CPU 分配分别为 33.3% 和 66.6%。需要注意的是，CPU 权重不代表绝对的 CPU 分配上限，如果 PHP 容器处于空闲状态，CPU 占用为 0，那么 Nginx 容器也允许使用 100% 的 CPU 资源。也就是说，`-c` 或 `--cpu-shares` 参数不是对 CPU 资源的硬性限制，只是限制实际允许的各容器对 CPU 资源的需求。

另外，还能够通过限制容器对 CPU 的时间占用，来限制 CPU 的使用。通过 `--cpu-period` 和 `--cpu-quota` 两个参数，就能实现限制 CPU 占用时间的目的。

```
$ sudo docker run -d --cpu-period 10000 --cpu-quota 5000 nginx:stable
```

其中，`--cpu-period` 参数表示计算调度的周期；`--cpu-quota` 参数表示在单一调度周期中，分配给这个容器的时间配额。此处 Nginx 容器最多可以得到一半的 CPU 时间配额。

## 2. 硬盘

硬盘是持久存放数据的主要场所，也是重要的计算机资源之一。同样的，如果硬盘中存在问题程序或者恶意程序，也会出现挤占硬盘 IO 导致其他容器受到影响的问题。特别是对于 IO 密集的应用，由于硬盘的带宽远不及 CPU 和内存，所以一旦这些程序出现问题，对其他容器造成的影响会高于 CPU 和内存带来的影响。

在 Docker 中，可以通过 `--device-read-bps` 和 `--device-write-bps` 命令限制指定硬盘的读写速度，也可以通过 `--device-read-iops` 和 `--device-write-iops` 命令限制 IO 数量。下面通过对比使用 IO 限制前和使用后的硬盘读写速度，来看看对 IO 的限制是否有效。

```
$ sudo docker run --rm ubuntu:latest dd if=/dev/zero of=/tmp/out bs=1M count=1024
oflag=direct
1024+0 records in
1024+0 records out
1073741824 bytes (1.1 GB) copied, 3.69373 s, 291 MB/s
$ sudo docker run --rm --device-read-bps /dev/sha:50mb ubuntu:latest dd if=/dev/zero
```



```
of=/tmp/out bs=1M count=1024 oflag=direct
1024+0 records in
1024+0 records out
1073741824 bytes (1.1 GB) copied, 20.4334 s, 52.5 MB/s
```

## 13.2.2 通过 ulimit 限制

在 Linux 系统中，可以通过 `ulimit` 命令对部分资源的使用进行限制。在每个容器中，也可以通过 `ulimit` 命令设置容器独立的限制和配置。当然，我们推荐通过创建容器时带入 `--ulimit` 参数来配置 `ulimit`，这样能够更直观地看到 `ulimit` 命令有了何种改变，也不需要再进入容器进行操作。

```
$ sudo docker run -d --name nginx --ulimit cpu=1000 nginx:stable
```

可以进入容器，查看对 `ulimit` 命令的配置有没有生效。

```
$ sudo docker exec -it nginx /bin/bash
root@985f2b112aac:/# ulimit -t
1000
```

通过 `ulimit` 命令可以修改 `Core dump` 文件大小、数据段大小、文件句柄数、进程栈深度、CPU 时间、单一用户进程数、进程虚拟内存等。另外，可以通过设置 `docker daemon` 中对 `ulimit` 的配置，配置容器的默认 `ulimit` 限制，因为所有容器的 `ulimit` 限制都继承于 `docker daemon` 中的默认配置。

```
$ sudo docker dockerd --default-ulimit cpu=1000
```

## 13.2.3 网络访问限制

对于网络的访问，也是保障 Docker 安全使用中需要特别注意的方面。可以将有着不同安全级别的容器分配在不同的网络中，特别是隐藏在内部可以信任的容器和暴露在外部网络可能被攻击的容器，以有效防止攻击。

例如，对外提供 Web 服务，既包括了暴露在外网中的 Nginx 容器，也包括了处理运行动态处理程序的容器等。通过数据卷容器来操作代码，即动态处理程序和 Nginx 都只能以只读的方式接触代码，就保障了即使 Nginx 容器被入侵，程序也不会被更改。但这样处理程序存在一个问题，如果被入侵的容器与部署代码的容器在宿主机中同属一个子网，那么攻击者就可能通过网桥实现入侵代码部署容器的目的，仍会影响系统的稳定性。

如图 13-2 所示, 为了保证安全, 可以为代码部署的容器与 Nginx 及用作包裹动态处理程序的容器分别配置不同的网络。这样就能避免攻击者通过宿主机的网桥入侵到安全级别不同、权限控制不同的容器中。

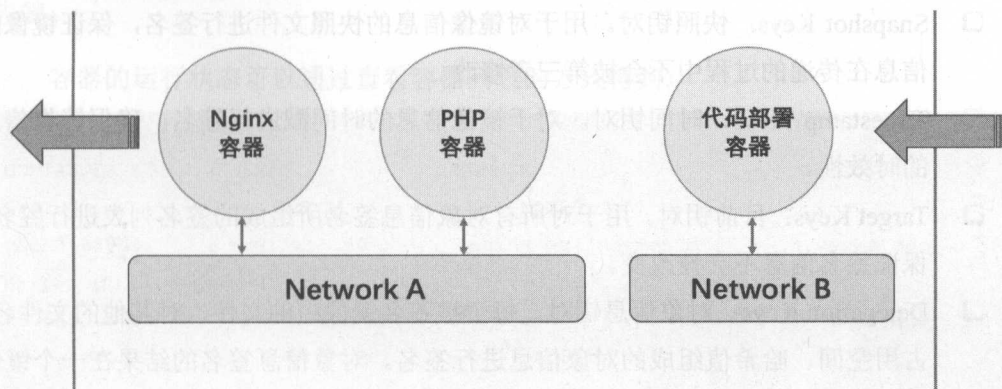


图 13-2 隔离不同功能的容器

如何使用和配置 Docker 中的网络, 我们在之前的章节中已经详细讲解过, 这里就不再展开介绍了。

## 13.3 校验与监控

校验数据的可靠性是安全防护的第一步, 也是最简单的一步。对于 Docker 这个可以通过互联网共享镜像等数据的程序来说, 数据校验也是极其关键的环节。而监控则能让我们实时了解 Docker 的运行情况, 帮助我们及时发现和解决问题。

### 13.3.1 镜像签名

可信升级框架 (The Update Framework) 是用于保证更新软件的完整性和安全性的框架。它类似于在 HTTPS 协议中所使用的证书签名机制, 对传递在互联网中的软件更新进行验证, 保证数据不在传输的过程中被篡改。Docker 也将可信升级框架使用到了镜像之中, 用来保证镜像在互联网中传递的完整性。

如图 13-3 所示, Docker 所采用的镜像验证由多层签名机制组成, 包含多个用于签名的公钥私钥对: Root Keys、Snapshot Keys、Timestamp Keys、Target Keys、Delegation Keys。

- ❑ **Root Keys:** 根钥对。用于签名 Snapshot、Timestamp、Target 的公钥等验证机制中的摘要信息。下载镜像的 Docker 程序，会通过根钥对中的私钥对签名来确保用于验证镜像信息的签名是否正确。由于根钥对是用于签名镜像所有摘要信息的，非常重要，所以 Docker 推荐通过离线等安全的方式传递根钥对。
- ❑ **Snapshot Keys:** 快照钥对。用于对镜像信息的快照文件进行签名，保证镜像的信息在传递的过程中不会被第三者修改。
- ❑ **Timestamp Keys:** 时间钥对。对于镜像信息的时间戳进行签名，确保镜像信息的时效性。
- ❑ **Target Keys:** 目标钥对。用于对所有对象信息签名所组成的签名列表进行签名，保证签名信息不会被改变。
- ❑ **Delegation Keys:** 对象信息钥对。每个被签名镜像中的文件，对其他的文件名、占用空间、哈希值组成的对象信息进行签名。对象信息签名的结果在一个镜像的签名中可以存在多个。

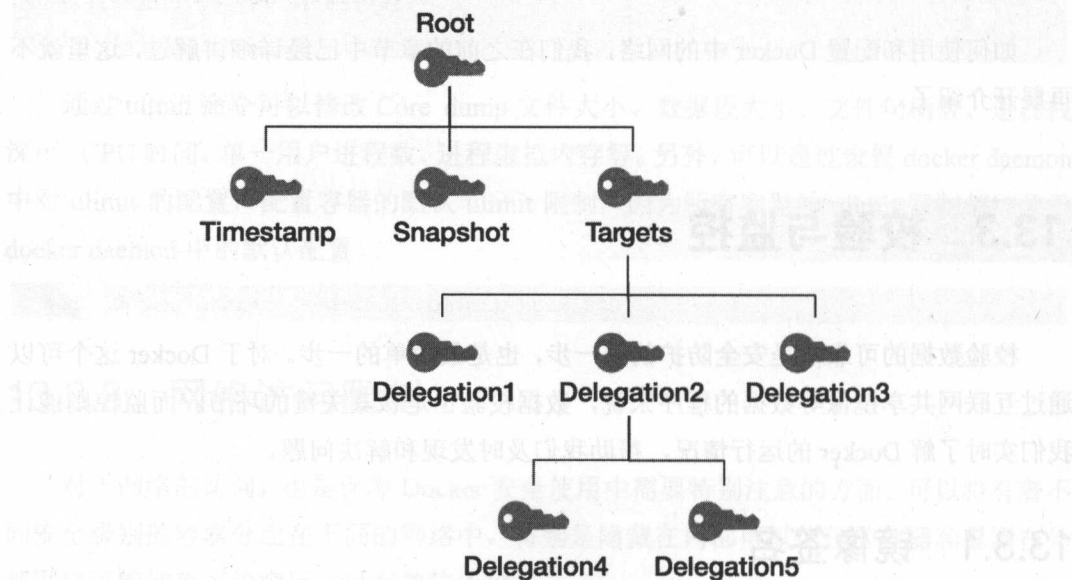


图 13-3 TUF 一致性签名关系链

当镜像的制作者将镜像推送到远程仓库时，Docker 会采用私钥对镜像进行签名，而当其他镜像使用者拉取这个镜像到本地时，Docker 会采用对应的公钥来验证此镜像与镜像制作者所发布的镜像是否一致。



## 13.3.2 运行状态监控

监控程序的运行状态是运行维护中最常见的工作，也是能够及时发现问题最有效的方法。监控的对象通常包括容器运行的状态、容器对资源的用量、程序代码中的状态报告等。

容器的运行状态可以通过查看容器列表的方式得到。

```
$ sudo docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED
caeb356806f1	nginx:1.10	"nginx -g 'daemon off'"	3 weeks ago
Exited (0) 3 weeks ago		nginx	
2896851da9de	php:7-fpm	"php-fpm"	3 weeks ago
Up 3 seconds	9000/tcp	phpfpm	
...			

如果容器的状态为 Up，则表示容器正在运行；如果容器的状态为 Exited，则说明容器已经停止。

容器对资源的占有可以通过 `docker stats` 命令来查看。这个命令能够显示容器所使用的 CPU 占用、内存消耗、IO 情况及进程数量等信息。

```
$ sudo docker stats --no-stream phpfpm
```

CONTAINER	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
phpfpm	0.01%	9.422 MiB / 1.953 GiB	0.47%	648 B / 648 B	0 B / 0 B	3

直接使用 `docker stats` 命令，Docker 会持续显示容器的状态，类似于 Linux 中的 `top` 命令。如果只显示输入状态时的容器状态，可以添加 `--no-stream` 参数。

在程序内部运行的一些信息，可以通过程序自己报告的情况来获取。在 Docker 中，通过 `docker logs` 命令就能显示出容器内主进程标准输出流输出的内容。

```
$ sudo docker logs nginx
```

```
2016/09/27 00:12:15 [emerg] 1#1: host not found in upstream "phpfpm" in /etc/nginx/fastcgi/phpfpm.conf:12
nginx: [emerg] host not found in upstream "phpfpm" in /etc/nginx/fastcgi/phpfpm.conf:12
```

在编写程序的时候，可以将需要的状态信息通过程序的标准输出流输出，这样在容器外部也可以得到这些信息了。

## 13.4 联级防护

除了独立使用 Docker 的安全机制来防范安全风险,还可以通过其他机制的配合形成更有利的安全防护网。

### 13.4.1 组合虚拟化

如果把 Docker 运行在宿主机的一套虚拟机系统中,就相当于让程序运行在 Docker 与虚拟机所构成的多套虚拟化体系中。这种容器配合虚拟机构成的虚拟化方案,我们称为组合虚拟化。

组合虚拟化相当于在 Docker 与宿主机系统中再增加一套防护,让宿主机系统更加安全,对于阻止攻击者进驻宿主机系统造成系统瘫痪,甚至数据泄露起到了非常积极的作用。

组合虚拟化的实现,可以采用 Docker 所提供的 Docker Machine 工具。Docker Machine 主要的功能在于为大多数较常见的操作系统提供 Docker 运行环境。由于 Docker 的运行依赖于 Linux 内核所提供的技术支持,所以使用 Docker 需要在 Linux 操作系统下。为了解决运行 Docker 的问题, Docker Machine 直接采用了虚拟机的方式在宿主机系统中搭建一个 Linux 运行环境,再在这个虚拟操作系统中搭建 Docker。

Docker Machine 本身是为了方便在更多的操作系统中使用 Docker 所设计的,但其虚拟机搭配 Docker 的设计正好符合了对搭建组合虚拟化的需求,所以可以使用它来实现组合虚拟化。Docker Machine 的使用方法,在之后的章节中会进行专门介绍。

### 13.4.2 文件系统安全

程序运行其实就是执行数据运算的过程,而数据绝大多数情况下是以文件的形式存储在硬盘之中的。对于文件系统来说,做到合理地安排目录和文件的权限,是防止恶意程序窃取敏感资料造成数据泄露,或者通过修改其他程序的执行文件导致系统瘫痪最简单、可靠的方式之一。

容器中的文件系统,是基于镜像所生产的一个可读写的 UFS 层。虽然在容器中修改

文件系统中的文件不会影响到镜像内的文件内容，但是如果攻击者潜入到容器中，仍然可以攻击这些沙盒环境中的文件，特别是容器中程序和所必需的依赖类库的文件。

通常情况下，容器中的大多数文件都是不需要进行写操作的，特别是对于运行的程序来说，我们可以在容器外部，也就是镜像中完成对它的修改，再以新容器的方式进行替换。得益于 Docker 容器秒级的启动速度，这个操作的难度和速度都不会超出我们的承受范围。

要让容器挂载只读的文件系统，只需要在创建容器时启用 `--read-only` 参数就能实现。下面来比较一下可读写挂载和只读挂载容器文件系统的区别。

```
$ sudo docker run -it --rm ubuntu /bin/bash
root@a2435eadf953:/# touch /demo.lock
root@a2435eadf953:/#

$ sudo docker run -it --rm --read-only ubuntu /bin/bash
root@3f8951900841:/# touch /demo.lock
touch: cannot touch '/demo.lock': Read-only file system
```

有时，我们会以数据卷的形式将容器中运行程序的配置挂载到容器内，实现容器与运行配置分离的目的。通常情况下，这些配置文件是不会在容器运行时被修改的，而一旦容器遭受攻击，就有可能造成配置被恶意攻击者篡改，从而导致容器内程序不能正常运行。

对于数据卷的挂载，我们同样可以控制这些文件和目录的访问权限。在创建容器并挂载数据卷时，只要在数据卷之后携带 `:ro`，就能实现数据卷的只读挂载，容器中的程序就丧失了对数据卷目录及文件的写权限。

```
$ sudo docker run -it --rm -v /ymdot/demo:/demo ubuntu /bin/bash
root@8cab2d0f4b1d:/# touch /demo/demo.lock
touch: cannot touch '/demo/demo.lock': Read-only file system
```

## 13.5 内核安全技术

在 Linux 中，还有许多提供 Linux 内核保护功能的机制和程序，保护 Linux 内核不被恶意篡改，更有利地防护容器中的虚拟操作系统以及宿主机操作系统的安全。



## 13.5.1 Capability

通常，运行在系统中的程序拥有操作所有系统接口的能力，也包括一些敏感的操作。例如，能够修改文件所有权限的 `chown` 命令，如果其他程序能够随意调用，就会对系统的稳定性造成不可预知的影响。

对于这类问题，Linux 提供了内核能力（Capability）机制。内核能力机制提供了控制程序对系统敏感操作权限的控制，是一种细粒度的权限控制方式。在内核能力机制中，系统的敏感操作被看作一个个内核能力，只有当程序拥有对应的内核能力的操作权限时，才能进行这些操作。

在 Linux 系统中，可以在 `/proc/<pid>/status` 信息文件中找到 `pid` 所对应进程的内核能力。

```
$ sudo cat /proc/240/status | grep Cap
CapInh: 0000000000000000
CapPrm: 0000001fffffffffff
CapEff: 0000001fffffffffff
CapBnd: 0000001fffffffffff
```

每个程序都带有 4 个 Capability 参数，它们都以十六进制的形式显示，每个数位都表示了一种内核能力的开关。

- ❑ **CapBnd (Boundary Capability)**: 系统能提供给程序内核能力的边界，表示了系统所有内核能力的数位范围。
- ❑ **CapPrm (Permitted Capability)**: 程序所能拥有的内核能力。
- ❑ **CapEff (Effective Capability)**: 程序实际拥有的内核能力，由于程序实际上并不一定申请使用所有的 `CapPrm`，所以会列出程序实际上会使用且允许使用的内核能力。
- ❑ **CapInh (Inheritable Capability)**: 当这个程序唤起和运行其他程序时，被唤起的程序能够得到的内核能力范围。

通过 Linux 的 Capability 机制，Docker 提供了容器级别的内核能力控制。也就是说，我们不但能够控制每个程序的内核能力使用权限，也能够对容器的内核能力权限进行控制。操作容器的内核权限时，其实就相当于对容器内虚拟系统的 `CapBnd` 进行了操作，所以容器中所有的程序，都不能逃逸出容器所定义的内核能力的范围。在 Docker 中以这种方式控制内核能力，不但能够与 Docker 的配置融合，方便在不同的主机间复用，也能够利用 Docker 容器为单一程序而生的特性，很容易地把对容器内核能力使用的控制，视为容器中程序对内核能力调用的控制。

Docker 为容器提供了内核能力白名单机制，只有在白名单中的能力才能被容器中的程序使用。在默认情况下，新创建的容器会携带一些最基本且相对比较安全的内核能力，可以在创建容器时携带 `--cap-add` 和 `--cap-drop` 两个参数，向白名单中添加或删除指定的内核能力。

修改文件所有者这个内核能力，默认存在于 Docker 容器的内核能力白名单中，如果不希望容器拥有这个能力，可以在创建容器时从容器的内核能力白名单中将这个内核能力移除。

```
$ sudo docker run -it --cap-drop chown debian:jessie /bin/bash
root@6c09759b9a9b:/# touch demo
root@6c09759b9a9b:/# chown www-data:www-data demo
chown: changing ownership of 'demo': Operation not permitted
```

我们看到，当修改文件所有者的能力移除后，即使容器是以根容器进行操作，也无法修改文件的所有者。

## 13.5.2 SELinux

无论是战场指挥，还是生产管理，权限机制都是保证安全的重要方法之一，在计算机系统中也不例外。

早期的操作系统没有考虑过安全隐患防护，也几乎没有权限控制的机制，任何人都可以随意访问计算机的数据和资源。随着安全需求的提高，大部分系统都引入了自主访问控制（Discretionary Access Control, DAC）机制。自主访问控制其实是指资源或数据的拥有者，控制其他人是否拥有操作和访问这些内容的权利。在实现中，其实就是系统中的用户，控制属于自己的资源是否能够被其他用户访问的机制。

虽然我们都以 DAC 进行系统中的权限控制，但 DAC 在进行安全保障方面仍有一定的缺陷。DAC 虽然能够限制其他用户对资源拥有者的资源的访问（也就是常说的客体对主体资源的访问），但其实，在 Linux 中，只要知道用户的密码（有时候甚至不需要知道密码），随时都可以通过 `su` 切换到对应的用户，这其中就包括了根用户。而根用户是众多系统关键文件的所有者，如果在这个用户下进行不当操作，容器很可能对系统的稳定性产生影响。

为了解决这个问题，就出现了强制访问控制（Mandatory Access Control, MAC）。强制访问控制将资源和数据的访问控制细粒化到具体的访问级别，它进行是否拥有访问权限的判断并不依赖于用户这种简单粗暴的体系，而是通过具体的安全策略进行。与

DAC 中程序可以通过指令修改资源或数据的访问权限不同，在 MAC 机制中，客体对主体的访问是固定的，不受程序的影响。

*SELinux (Security-Enhanced Linux)* 是美国国家安全局对 MAC 机制的一种实现，功能全面，而且其基于 MAC 机制多年的研究经验，并且进行了充分的测试，拥有相当高的安全系数。目前，包括 Fedora、Red Hat、Debian、Cent OS 在内的大部分 Linux 发行版本都内置了 SELinux 模块，用来保障系统核心文件的安全性。

在 Docker 中，同样可以利用 SELinux 来限制容器中虚拟操作系统内的资源和数据的访问性。在 Docker 中使用 SELinux，要先确定当前系统中的 SELinux 是否支持 Docker。通过 `semodule -l` 命令可以查看到 SELinux 所包含的模块信息，如果 SELinux 支持 Docker，则与 Docker 对应的模块也会出现在其中。

```
$ sudo semodule -l | grep docker
docker 1.0.0
```

如果结果中出现了 `docker`，则表示系统中的 SELinux 是支持 Docker 的。此处结果中的 1.0.0，就是 SELinux 的 Docker 模块的版本。

如果要在 Docker 服务中开启 SELinux 的支持，可以在启动 Docker daemon 时携带 `--selinux-enabled` 参数。

```
$ sudo docker daemon --selinux-enabled=true
```

在较新版本的 Docker 中，Docker daemon 的启动命令已经改为 `dockerd` 命令，其使用方法如下。

```
$ sudo dockerd --selinux-enabled=true
```

### 13.5.3 AppArmor

与 SELinux 相似，AppArmor 也是一个强制访问控制方案的实现。相对于 SELinux 来说，AppArmor 得益于高效和易用等特性，在 Linux 中得到了广泛使用。AppArmor 能够对操作系统和应用程序进行从内而外的保护，即使基于 0Day 这类危险漏洞的攻击，AppArmor 也可以进行防护。

在 Docker 中，AppArmor 得到了相对强大的支持。在 Docker daemon 启动时，只要检查到 Linux 内核中支持 AppArmor，就会为 Docker 创建 AppArmor 的配置文件，并应用这个配置文件。



用于 Docker 的 AppArmor 配置位于 `/etc/apparmor.d/docker` 文件中，我们可以修改这个配置文件来控制在 Docker 中使用 AppArmor 的规则。

如果要将其他的 AppArmor 配置用于 Docker 容器中，可以在容器启动时通过 `--security-opt` 参数选择容器所使用的 AppArmor 配置。

```
$ sudo docker run -d --security-opt apparmor:nginx nginx
```

为了保障 Docker 的运行不会影响到宿主机系统的安全，在实际投入生产的 Docker 服务中，建议同时开启 SELinux 和 AppArmor。

## 13.6 本章小结

在本章中，我们深入浅出地对能够帮助 Docker 提升安全性的系统内核机制、控制参数配置等进行了说明。同时，也对 Docker 在本身设计中加强安全的措施有了一定的了解。对于能够增加 Docker 安全防护能力的软件和工具，也做了简单介绍。

安全是一个老生常谈的问题，做好安全防护是任何一个系统都必不可少的工作，也是开发和运维人员必须要掌握的基础知识。

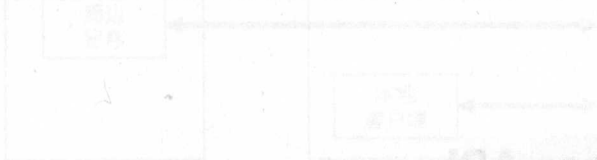


图 14-1 Docker API 接口

### 14.1.2 关于 RESTful

Docker 能够广泛发展的原因之一，就是拥有开放的精神。世界是开放的，技术也是开放的。Docker 不仅是一个容器引擎，更是一个开放的平台。它通过 RESTful API 提供了与容器引擎的交互接口，使得开发者可以方便地集成 Docker 到自己的应用中。这种开放的精神，使得 Docker 能够快速地被社区接受，并得到了广泛的应用。

# 第 14 章

## Docker API

Docker 作为部署和运行程序的容器技术提供者，自然需要对它进行控制和监控。在前面的章节中，我们都是使用 `docker` 命令对 Docker 下达指令。其实在 Docker 所提供的操作方式里，只有一种能够操作 Docker 的方法——Docker API。包括 `docker` 命令程序在内的所有能够对 Docker 下达指令或者能够从 Docker 中获取信息的工具，其实都是使用 Docker API 来实现的。在本章中，我们会专门介绍 Docker API，这个囊括了所有控制 Docker 方法的接口。

### 14.1 关于 Docker API

Docker API 也是 Docker 的精髓之一，通过 Docker API 这一统一的接口，我们不但能够完全操控 Docker，也能根据需求开发出自己的 Docker 管理工具。

#### 14.1.1 通用操作接口

Docker 能够飞速发展的原因之一，就是拥有开放的精神。世界各地的开发者贡献的代码，不但让 Docker 得到了不断地改进，也让 Docker 周边软件所组成的体系日渐成熟。如果把 Docker 比作航母，那围绕在 Docker 周围的软件或插件就是它的护卫舰。这支庞大的航母战斗群的力量是不容小觑的。

大部分 Docker 周边的软件，都以辅助 Docker 为目的，所以它们需要对 Docker 中的容器及其他组件进行操作。这些操作如果都通过 Docker 的命令行程序 Docker CLI 来完成，是存在一定局限性的。这些局限性不但表现在操作指令需要通过 Docker CLI 中转，才能作用到 Docker 服务程序中；而且表现在对远程进行的 Docker 操作，使用 Docker CLI 还需要搭配具有远程 Shell 功能的软件。这种限制无疑会阻碍周边软件的发展。于是，Docker 提供了另一种更方便、更通用的方法来解决服务调用的问题，即 Docker API。

Docker API 是一套基于 HTTP，用于操作 Docker 服务的接口。它实现在 Docker 服务程序中，也由 Docker 服务程序向外提供。所以说，使用 Docker API 其实操作的都是 Docker 服务本身，不会经由其他程序中转。

如图 14-1 所示，Docker API 将对 Docker 的操作封装成了一个 HTTP 接口，只要根据相应的规则对这些接口进行调用，就能直接对 Docker 中的内容进行操作，并获取 Docker 的运行状态等信息。由于 HTTP 本身就是依赖于网络的，所以通过 HTTP 实现的接口就具有远程操作的能力，这就使编写的程序能够直接在另一台主机上向 Docker 下达指令。

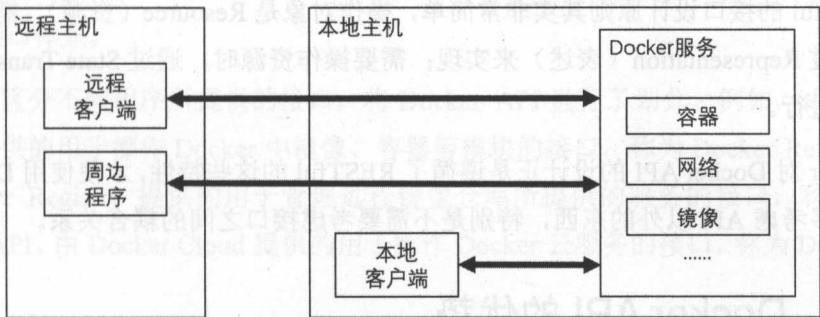


图 14-1 Docker API 连接程序与 Docker 服务

Docker API 利用 HTTP 简单易用，并且已经广泛普及的优势，让使用不同编程语言的开发者都能很快地进行基于 Docker 的周边软件开发。我们熟知的 docker 命令，其实也是通过 Docker API 来控制 Docker 服务的。

### 14.1.2 关于 RESTful

Docker API 是基于 RESTful 设计的 HTTP 接口。RESTful (Representational State Transfer, 表述性状态转移) 是一套简单、清晰、低耦合地面向资源的 HTTP 接口设计规范，充分使用 HTTP 面向资源、无状态等特性解决了接口之间耦合性的问题。同时，使



用统一资源定位的语义，使 RESTful 的接口简单易读也易懂。在很多情况下，甚至都不需要文档，只要通过 RESTful 接口的地址，就能知道接口的功能与作用。

在 RESTful 设计中有以下几个比较关键的概念。

- **Resource:** 资源。RESTful 接口操作或获取的对象就是资源，在 Docker 里就是容器、镜像、数据卷等实体。所以说，采用 RESTful 设计的 Docker API，其实就是为我们提供了操作 Docker 中的容器、镜像、数据卷等模块的方法。
- **Representation:** 表述。因为资源的形式各式各样，有的甚至是二进制文件，所以我们不能直接展示它们。在 RESTful 设计中，要求采用可读性的格式去展示资源，虽然展示的也许并非资源本身，但却能够充分表述资源。在 Docker API 中，大都以 JSON 文本的形式表述资源。
- **State Transfer:** 状态转移。对于资源的修改可以理解为资源状态发生的变化，也就是状态转移。在 RESTful 设计中主张使用 HTTP 中的方法确定资源的操作方式。例如，使用 GET 方法获取资源，利用 POST 方法新增资源，利用 PUT 方法修改资源，利用 DELETE 方法删除资源等。

RESTful 的接口设计原则其实非常简单，操作对象是 Resource（资源），需要展示资源时则通过 Representation（表述）来实现；需要操作资源时，通过 State Transfer（状态转移）来进行。

Docker 对 Docker API 的设计正是遵循了 RESTful 的这些特性，才使使用 Docker API 时无须过多考虑 API 以外的东西，特别是不需要考虑接口之间的耦合关系。

### 14.1.3 Docker API 的优势

Docker API 采用遵循 RESTful 规范的 HTTP 接口实现，较其他软件接口有非常多的优势，它的轻量性、普及性、标准性和易维护性都远超普通的程序接口设计。

因为使用 HTTP 作为接口的载体，所以可以很轻松地连接和使用 Docker API 操作 Docker，不需要使用特别的客户端，也不需要编写相关的通信程序。这种轻量级的设计，使用户可以更容易地操作 Docker。在没有 Docker 客户端程序的情况下，依然可以根据 Docker API 通过 HTTP 使用 Docker。

Docker API 使用 HTTP 进行数据传输，在请求过程中，请求参数以符合 RESTful 的形式传递到 Docker 服务程序中，返回的结果大多以 JSON 的形式输出。HTTP、RESTful 规范、JSON 格式等 Docker API 所涉及的基础架构，都是已经被广泛使用的标准或规范，

也是很容易被掌握的内容。所以，我们不需要花费太多的时间在 Docker API 上，基本都能够做到拿来即用。

我们熟知，Docker 由许多模块组成，也有很多周边配套程序，对于这些模块或者程序，Docker 官方都以 RESTful 接口的形式提供操作方法，这就大幅降低了学习和掌握多套接口规范及使用方法的成本。

采用简单通用的标准来设计 Docker API，使 Docker API 不但简单易懂，而且非常自由和便于开发。不论是维护原有的接口和推出新的接口的 Docker 官方人员，还是与之对接的程序开发者，都能很快地完成工作。

### 14.1.4 Docker API 的分类

之前我们提到了 Docker 服务程序提供了一套 HTTP 接口，用于给使用者提供操作 Docker 的统一方法。这是 Docker 官方开发者做出的良好设计，可以在开发 Docker 周边服务或程序的同时，采用 HTTP 和 RESTful 规范来设计和实现各自的接口，如 Docker Registry 镜像仓库程序、Docker Cloud 云服务，Docker API 就是由这些接口共同组成的 Docker 接口体系。

为了区分不同程序所提供的接口，将 Docker API 进行了划分。例如，由 Docker 服务程序提供的用于操作 Docker 中镜像、容器等模块的接口，称为 Docker Remote API。由 Docker Registry 提供的用于管理远程镜像仓库所提供的服务的接口，称为 Docker Registry API。由 Docker Cloud 提供的用于操作 Docker 云服务的接口，称为 Docker Cloud API。

我们最常用的 Docker API 体系中的接口应该是 Docker Remote API 和 Docker Registry API 两种。下面就着重对这两种 Docker API 进行讲解、举例和实践。

## 14.2 使用 Docker Remote API

Docker Remote API 是 Docker API 中最基础的部分，它的作用就是控制 Docker 的核心服务——Docker daemon。了解 Docker Remote API 不但可以让我们熟悉这些 API 的使用，也能够让我们更好地理解对 Docker 下达的命令是如何控制 Docker 工作的。

## 14.2.1 关于 Docker Remote API

Docker Remote API 是 Docker API 中最重要的部分，它能够控制 Docker 服务及其中的容器、镜像、网络等功能的运行。Docker Remote API 是由 Docker 服务程序提供的，会随着 Docker 服务的运行而启动。在默认的配置中，Docker Remote API 监听来自本地的连接，连接方式是 UNIX Socket，监听地址位于 `unix:///var/run/docker.sock`，可以通过 Docker daemon 中的 `-H` 或 `--host` 参数修改和覆盖默认的监听地址。

基本所有的 Docker Remote API 都是遵循 RESTful 设计规范的，除了少数需要进行长时间操作的接口，如连接到容器、拉取镜像等。因为这几个接口需要以长连接的方式长时间地从 Docker daemon 中获取容器中的输出或下载镜像时的进度，所以它们并不完全遵循 RESTful 规则。对于这些接口，我们需要连接 HTTP 的输出流，再从输出流中获得数据。

我们通过 cURL 工具，从本地通过 UNIX Socket 请求一个简单的 Docker Remote API。请求的接口地址为 `/info`，这是一个能够获得 Docker 运行的宿主机环境的接口。

```
$ sudo curl --unix-socket /var/run/docker.sock http://localhost/info
{
  "ID": "SOTT:ZZU6:FKDT:URCE:BKLS:Y6ZF:EQYZ:WMJ3:ZSDH:34MX:2GRP:KWYM",
  "Containers": 10,
  "ContainersRunning": 5,
  "ContainersPaused": 0,
  "ContainersStopped": 5,
  "Images": 50,
  "Driver": "devicemapper",
  "DriverStatus": [
    [
      "Pool Name",
      "docker-202:1-1452214-pool"
    ],
    [
      "Pool Blocksize",
      "65.54 kB"
    ],
    [
      "Base Device Size",
      "10.74 GB"
    ]
  ]
}
```



```
    "Backing Filesystem",
    "xfs"
  ],
  [
    "Data file",
    "/dev/loop0"
  ],
  [
    "Metadata file",
    "/dev/loop1"
  ],
  [
    "Data Space Used",
    "2.194 GB"
  ],
  [
    "Data Space Total",
    "107.4 GB"
  ],
  [
    "Data Space Available",
    "36.57 GB"
  ],
  [
    "Metadata Space Used",
    "3.899 MB"
  ],
  [
    "Metadata Space Total",
    "2.147 GB"
  ],
  [
    "Metadata Space Available",
    "2.144 GB"
  ],
  [
    "Thin Pool Minimum Free Space",
    "10.74 GB"
  ],
  [
    "Udev Sync Supported",
    "true"
```

```
],
[
    "Deferred Removal Enabled",
    "false"
],
[
    "Deferred Deletion Enabled",
    "false"
],
[
    "Deferred Deleted Device Count",
    "0"
],
[
    "Data loop file",
    "/var/lib/docker/devicemapper/devicemapper/data"
],
[
    "Metadata loop file",
    "/var/lib/docker/devicemapper/devicemapper/metadata"
],
[
    "Library Version",
    "1.02.107-RHEL7 (2016-06-09)"
]
],
"SystemStatus": null,
"Plugins": {
    "Volume": [
        "local"
    ],
    "Network": [
        "null",
        "host",
        "overlay",
        "bridge"
    ],
    "Authorization": null
},
"MemoryLimit": true,
"SwapLimit": true,
"KernelMemory": true,
```



```

"CpuCfsPeriod": true,
"CpuCfsQuota": true,
"CPUShares": true,
"CPUSet": true,
"IPv4Forwarding": true,
"BridgeNfIptables": true,
"BridgeNfIp6tables": true,
"Debug": false,
"NFD": 46,
"OomKillDisable": true,
"NGoroutines": 63,
"SystemTime": "2016-10-09T08:03:41.274621867+08:00",
"ExecutionDriver": "",
"LoggingDriver": "json-file",
"CgroupDriver": "cgroupfs",
"NEventsListener": 0,
"KernelVersion": "3.10.0-123.9.3.el7.x86_64",
"OperatingSystem": "CentOS Linux 7 (Core)",
"OSType": "linux",
"Architecture": "x86_64",
"IndexServerAddress": "https://index.docker.io/v1/",
"RegistryConfig": {
  "InsecureRegistryCIDRs": [
    "127.0.0.0/8"
  ],
  "IndexConfigs": {
    "docker.io": {
      "Name": "docker.io",
      "Mirrors": null,
      "Secure": true,
      "Official": true
    }
  },
  "Mirrors": null
},
"NCPU": 1,
"MemTotal": 1929199616,
"DockerRootDir": "/var/lib/docker",
"HttpProxy": "",
"HttpsProxy": "",
"NoProxy": "",
"Name": "iz23ztkpmckZ",

```



```
"Labels": null,
"ExperimentalBuild": false,
"ServerVersion": "1.12.1",
"ClusterStore": "",
"ClusterAdvertise": "",
"SecurityOptions": [
    "seccomp"
],
"Runtimes": {
    "runc": {
        "path": "docker-runc"
    }
},
"DefaultRuntime": "runc",
"Swarm": {
    "NodeID": "",
    "NodeAddr": "",
    "LocalNodeState": "inactive",
    "ControlAvailable": false,
    "Error": "",
    "RemoteManagers": null,
    "Nodes": 0,
    "Managers": 0,
    "Cluster": {
        "ID": "",
        "Version": {},
        "CreatedAt": "0001-01-01T00:00:00Z",
        "UpdatedAt": "0001-01-01T00:00:00Z",
        "Spec": {
            "Orchestration": {},
            "Raft": {},
            "Dispatcher": {},
            "CAConfig": {},
            "TaskDefaults": {}
        }
    }
},
"LiveRestoreEnabled": false
}
```

由于请求和处理都在本地发生，所以很快就能得到 Docker Remote API 返回的结果。接口以 JSON 形式返回想要的信息。对于/info 这个接口，请求后可以得到 Docker 的镜像

和容器等模块的数据、远程镜像仓库的配置等信息。这里只是对使用 Docker Remote API 进行简单说明，返回结果中条目的内容将在后面的小节进行详细解读。

注意：Docker API 接口所返回的 JSON 格式，都是没有空格和换行的紧密格式，本书为了更好地向读者展示 Docker API 所返回的结果，将这些 JSON 进行了格式化。

Docker Remote API 是以 HTTP 形式进行请求的，上面展示的是通过 UNIX Socket 进行的本地请求，远程请求可以直接通过服务器地址进行。这里同样以 cURL 程序作为客户端。

```
$ sudo curl http://<host>:<port>/info
```

port 对应了在 Docker daemon 中所设置的 Docker Remote API 的监听端口，host 则是 Docker 宿主机的 IP 地址。

### 14.2.2 Docker Remote API 的版本

Docker Remote API 随着 Docker 的迭代也在不断更新，Docker 版本与 Docker Remote API 版本的对应关系如表 14-1 所示。

表 14-1 Docker与Docker Remote API版本的对应关系

Docker版本	Docker Remote API版本
1.12.x	1.24
1.11.x	1.23
1.10.x	1.22
1.9.x	1.21
1.8.x	1.20
1.7.x	1.19
1.6.x	1.18

使用 Docker Remote API 时可以携带版本号，即在接口路径前加上表示版本的 v 及对应的版本。

```
/v1.24/
```

例如，上面示例中的/info 接口，也可以通过/v1.24/info 来访问它。Docker 对 Remote

API 做了兼容适配，即使用接口版本为 1.24 的 Docker daemon 时，也可以通过/v1.23/info 来访问旧的 Docker Remote API 并获取对应的信息。所以如果需要进行基于 Docker Remote API 的开发，那么最好通过指定的版本号来访问 Docker Remote API，这样能最大限度地避免因为 Docker daemon 中 Docker Remote API 版本的改变引起的连锁反应。当然，这种兼容适配只作用于较新的几个 Docker daemon 和 Docker Remote API 版本中，所以仍然需要及时更新迭代来完成新接口的对接。

14.2.3 通过 Remote API 列出容器

除了通过 docker ps 命令列出 Docker daemon 中的容器，还可以通过 Docker Remote API 来获取这些信息。列出容器的 API 接口地址为：

```
GET /containers/json
```

在请求列出容器的接口时，传入的参数及说明如表 14-2 所示。

表 14-2 传入的参数及说明

查询参数	参 考 值	参数说明
all	0, 1	是否返回所有容器，即使容器不在运行状态
limit	10	限制输出容器的数量
since	9cd87474be90	从指定容器ID之后开始返回
before	9cd87474be90	从指定容器ID之前开始返回
size	0, 1	是否显示容器的大小
filters	exited=1	根据规则过滤容器

列出容器接口返回参数的样例：

```
[
  {
    "Id": "9cd87474be90",
    "Names": ["/coolName"],
    "Image": "ubuntu:latest",
    "ImageID": "d74508fb6632491cea586a1fd7d748dfc5274cd6fdfedee309ecdcbc2bf5cb82",
    "Command": "echo 222222",
    "Created": 1367854155,
    "State": "Exited",
```



$$\left. \begin{array}{l} \} , \\ \{ \end{array} \right\}$$

```
    "HostConfig": {
      "NetworkMode": "default"
    },
    "NetworkSettings": {
      "Networks": {
        "bridge": {
          "IPAMConfig": null,
          "Links": null,
          "Aliases": null,
          "NetworkID": "7ea29fc1412292a2d7bba362f9253545fecdfa8ce9a6e37dd10ba8bee7129812",
          "EndpointID": "d91c7b2f0644403d7ef3095985ea0e2370325cd2332ff3a3225c4247328e66e9",
          "Gateway": "172.17.0.1",
          "IPAddress": "172.17.0.5",
          "IPPrefixLen": 16,
          "IPv6Gateway": "",
          "GlobalIPv6Address": "",
          "GlobalIPv6PrefixLen": 0,
          "MacAddress": "02:42:ac:11:00:05"
        }
      }
    },
    "Mounts": []
  }
}
```

14.2.4 通过 Remote API 列出镜像

Docker daemon 本地镜像仓库中的镜像可以通过 Docker Remote API 来获取。

```
GET /images/json
```

在请求列出镜像的接口时，传入的参数及说明如表 14-3 所示。

表 14-3 传入的参数及说明

查询参数	参 考 值	参数说明
all	0, 1	是否返回所有镜像
filters	dangling=true	根据规则过滤镜像
filter	mysql	过滤返回数据

列出镜像接口返回参数的样例：

```
[
  {
    "RepoTags": [
      "ubuntu:12.04",
      "ubuntu:precise",
      "ubuntu:latest"
    ],
    "Id": "8dbd9e392a964056420e5d58ca5cc376ef18e2de93b5cc90e868a1bbc8318c1c",
    "Created": 1365714795,
    "Size": 131506275,
    "VirtualSize": 131506275,
    "Labels": {}
  },
  {
    "RepoTags": [
      "ubuntu:12.10",
      "ubuntu:quantal"
    ],
    "ParentId": "27cf784147099545",
    "Id": "b750fe79269d2ec9a3c593ef05b4332b1d1a02a62b4accb2c21d589ff2f5f2dc",
    "Created": 1364102658,
    "Size": 24653,
    "VirtualSize": 180116135,
    "Labels": {
      "com.example.version": "v1"
    }
  }
]
```

## 14.3 使用 Docker Registry API

Docker Registry API 可以管理和使用远程镜像仓库，即 Docker Registry，了解 Docker Registry API 就可以更熟悉 Docker 与远程仓库交互的流程和方式。



### 14.3.1 关于 Docker Registry API

Docker Registry API 专门用于管理 Docker Registry 及其中所包含镜像的接口，由 Docker Registry 提供。

我们知道，Docker Registry 是由 Docker 提供的一个远程镜像服务，已经被封装成了镜像，只要通过 registry 这个镜像名称，就能在 Docker Hub 中找到它。我们能够通过 Docker Registry 搭建私有的远程镜像仓库，自然也需要有方法去操作它，Docker Registry API 就是这个能够操作 Docker Registry 的方法。

通常，使用远程镜像仓库是为了在不同的主机间传递和共享镜像。所以，最常用的方法是对镜像的推送和拉取。

由于镜像往往具有一定的体积，即使 Docker 对单个镜像进行了分层处理，但每个镜像层仍然占据不少的体积，而体积较大的内容通过 HTTP 直接进行传输效果并不太理想。由于可能存在网络异常，这就会导致传递的数据因为不全而被丢弃，也会导致 HTTP 连接断开的情况。

Docker Registry API 通过拆分镜像数据，分别进行数据的传送，减少了因网络问题导致的需要重传所有镜像数据的情况。

### 14.3.2 Docker Registry API 的主要功能

Docker Registry 是用于存储和管理镜像的仓库，所以自然少不了对镜像最基础的增删改查等操作。同时，能够操作 Docker Registry 的 Docker Registry API，自然少不了对镜像的操作。而且，通过 Docker Registry API 还能够直接或间接地得到许多功能上或体验上的提升。Docker Registry API 主要提供了以下几个功能。

- ❑ 镜像信息操作：镜像信息指的是镜像的 ID、仓库名、标签、启动命令等能够全面描述镜像的内容。包括了所有组成镜像的镜像层数据，以及能够从仓库中识别出它们的必要信息。在 Docker Registry API 中，不论是想要推送镜像还是拉取镜像，都需要先进行镜像信息的推送和拉取，这样才能准确地将每个镜像层上传到服务器或下载到本地。
- ❑ 镜像验证：对于镜像的可靠性，我们完全可以通过 Docker Registry API 所给出的镜像信息中相关的字段进行校验。除了比对镜像的信息在传递的过程中是否得到修改，还可以通过签名字段判断镜像的数据是否被中间者攻击。这对于保障镜像的安全、完整是非常重要的环节。

- ❑ 镜像推送: Docker Registry API 提供了分块方式将镜像数据推送到 Docker Registry 服务器。这种方式不但能够让客户端同时上传多个数据块, 还能减少因为某块数据传输失败引起其他数据块传输失败的问题。Docker Registry 收到上传的镜像数据后, 会先进行数据的组装, 在调用 Docker Registry API 完成镜像上传时, 缺失的数据会得到 Docker Registry 的通知, 这时就可以补充上传未完成的数据。
- ❑ 镜像拉取: 与镜像推送类似, Docker Registry API 也把拉取镜像的过程设置为拉取镜像的数据块, 再到本地进行镜像数据的组装。通过这种方式, 可以在拉取时并发进行多个数据块的传输, 并且减少网络异常引起的问题, 提高镜像传输效率。
- ❑ 镜像层控制: Docker Registry API 做到了针对镜像层数据的控制, 当我们推送一个镜像层到 Docker Registry 时, Docker Registry API 会先通过传递的镜像散列值, 判断这个镜像层是否已经存在于 Docker Registry 中。对于已经存在于 Docker Registry 中的镜像层, 我们会在 Docker Registry API 的响应内容中得到提示, 这时就无须传递这些重复的数据, 减少了传递数据的大小, 提高了推送数据的时间。

### 14.3.3 Docker Registry API 的版本

与 Docker Remote API 类似, Docker Registry API 也有版本机制, 以保障在迭代过程中降低与其他程序之间的耦合。

目前, Docker Registry 的最新版本为 2.5.1, 而 Docker Registry API 对应的 Docker Registry 的主版本号就是 2。Docker Registry API 的第 2 个版本比第 1 个版本有了很多提升, 所以建议直接使用最新版本。由于这个 Docker Registry 使用的是 HTTP, 所以最新版本的接口全称为 Docker Registry HTTP API V2。

如果要知道 Docker Registry 是否支持某个版本的 API, 可以通过下面的请求来进行判断。

```
GET /{version}/
```

version 代表需要判断的版本。例如, 要判断 Docker Registry 是否支持 Docker Registry HTTP API V2, 则可以请求/v2/这个地址。

如果 Docker Registry 返回的 HTTP 响应码为 200, 则表示请求是成功的。如果响应

码为 401，则表示请求需要授权，即需要提供身份凭证。若响应码为 200 或 401，则可以认为请求的 Docker Registry 是支持给出版本的 Docker Registry API 的。如果请求的返回结果是 404，则表示 Docker Registry 不支持给出的 API 版本。

同时，最新的 Docker Registry 也已经在开发周期中了，预计在发布之时，还将有常量镜像引用等新的特性加入 Docker Registry 之中。届时 Docker Registry API 也会随 Docker Registry 的更新而迭代到新的版本。

### 14.3.4 通过 Registry API 拉取镜像

从 Docker Registry 中拉取镜像是使用 Docker Registry 最常见的操作之一。在拉取镜像之前，先通过 Docker Registry API 从 Docker Registry 中获得想要拉取镜像的信息。获取镜像信息的 API 地址为：

```
GET /v2/<name>/manifests/<reference>
```

name 表示镜像在远程仓库中的名称，reference 可以是镜像的标签或摘要串。通过这两个参数，就可以让 Docker Registry API 识别出远程镜像仓库中唯一的镜像，并将它的信息返回给我们。

如果给出的镜像名称和标签在远程仓库中没有对应的镜像存在，请求的获取镜像信息的接口会返回 404 这个 HTTP 错误响应码。如果镜像存在，镜像信息会以 JSON 的形式被返回到响应体中。在所有返回的信息中，比较重要的有以下几项。

- ❑ name: 镜像名称。
- ❑ tag: 镜像标签。
- ❑ fsLayers: 所有镜像层的信息。
- ❑ signature: 镜像的签名信息。

得到镜像的信息后就可以拉取镜像的内容了。在镜像信息的 fsLayers 字段中，包含了所有镜像内所有镜像层的信息，这里需要采用 blobSum 进行镜像层的拉取。

在 Docker Registry 中，每个镜像层都会被独立存储，这样就能保证镜像与镜像之间可以共享镜像层。我们通常以镜像层的散列值来识别镜像层，对于镜像信息中的镜像层来说，散列值就是 blobSum 字段。

拉取镜像层的 Docker Registry API 接口为：

```
GET /v2/<name>/blobs/<digest>
```



`name` 表示镜像的仓库名, `digest` 表示需要下载的镜像层的散列值, 也就是镜像信息中镜像层的 `blobSum` 字段。拉取镜像的接口是识别镜像层时通过 `digest` 来完成的, `name` 字段只用作标记效果。

拉取镜像层的过程和普通的 HTTP 请求一致, 只是数据可能相对普通的请求要多, 所以需要更多的时间来完成。在下载镜像层的请求过程中, 我们可能会收到 307 这个 HTTP 响应码, 这表示 Docker Registry 引导我们转到外部地址下载这个镜像层。另外, 下载镜像层的过程也能使用 HTTP 的缓存设计, 也就是说, E-Tag、If-Modified-Since 这些 HTTP 请求头, 都可以使用在下载镜像层的过程中, 以提高下载速度。

所有的镜像层下载完成后, 由它们组成的镜像就存在于本地主机中了。

### 14.3.5 通过 Registry API 推送镜像

推送镜像类似于拉取镜像的逆过程, 先将所有的镜像层上传到 Docker Registry, 再把镜像信息传给服务器。

要推送镜像需要先请求下面这条 API 来判断 Docker Registry 是否已经准备好接收镜像的上传。

```
POST /v2/<name>/blobs/uploads/
```

如果这条 API 返回正常, 就可以开始上传镜像。这条 API 的响应头中存在一个 `UUID` 字段, 在上传镜像的过程中, 需要使用它将所有上传及其他接口调用的过程归入同一个上传流程中。

上传镜像与下载镜像一样, 也是将镜像分为镜像层, 对单个镜像层进行上传。上传每个镜像层之前, 先判断这个镜像层是否存在于远程镜像仓库中, 如果存在, 则无须上传这个镜像层, 以减少上传的数据量和上传时间。判断镜像层是否存在于远程仓库中, 可以通过下面这条 Docker Registry API 来实现。

```
HEAD /v2/<name>/blobs/<digest>
```

我们需要传给这个 API 的参数为: 镜像的名称与镜像层的散列值, Docker Registry 主要根据镜像层散列值来判断镜像层是否已经存在于仓库中。如果镜像层已经存在于远程仓库, 接口会响应 200 响应码, 我们就没有必要再上传这个镜像层了。不存在于远程镜像仓库中的镜像, 可以通过上传接口进行上传。Docker Registry API 提供了两个上传镜像的接口, 分别用于整体上传和分块上传。

整体上传的请求路径和需要传递的请求头为：

```
PUT /v2/<name>/blobs/uploads/<uuid>?digest=<digest>
Content-Length: <size of layer>
Content-Type: application/octet-stream
```

分块上传的请求路径和需要传递的请求头为：

```
PATCH /v2/<name>/blobs/uploads/<uuid>
Content-Length: <size of chunk>
Content-Range: <start of range>-<end of range>
Content-Type: application/octet-stream
```

如果使用分块上传，在上传最后一个块时，也需要使用整体上传的接口完成镜像层的上传。

当所有镜像层上传完成时，需要将镜像信息传递给 Docker Registry，告知它镜像的属性，以及所有组成镜像的镜像层信息。提交镜像信息的接口和请求参数的示例如下：

```
PUT /v2/<name>/manifests/<reference>
{
  "name": <name>,
  "tag": <tag>,
  "fsLayers": [
    {
      "blobSum": <digest>
    },
    ...
  ],
  "history": <v1 images>,
  "signature": <JWS>,
  ...
}
```

当然，除了完成镜像层的上传，我们也可以取消镜像的上传。在取消镜像上传时，最好调用取消上传的接口，告知 Docker Registry 这次上传已经取消，可以释放与之相关的资源。

```
DELETE /v2/<name>/blobs/uploads/<uuid>
```

调用取消接口之后，上传的 UUID 就会失效，其他对此次上传的调用也不会完成。

## 15.1.1 Docker Compose 简介

## 14.4 本章小结

Docker API 是 Docker 官方开发人员优秀的设计, 这种设计完全规范和统一了 Docker 及相关软件的操作方式。同时, 采用 HTTP 接口的形式实现的 Docker API, 能够兼容几乎所有的编程语言, 也支持在网络中传递指令。

掌握好 Docker API 是使用 Docker, 特别是需要开发整合 Docker 或管理 Docker 中的项目程序时最重要的技能之一。通过本章的讲解, 我们对 Docker API 的基本架构及简单的使用方法有了一定的了解。Docker API 设计清晰, 规范简单, 我们可以很快地通过对本章知识的学习, 举一反三, 对其他未提及的 Docker API 快速上手。



# 第 15 章

## 管理工具

我们之前一直讲解的 Docker，准确来说应该是 Docker Engine，即包含了提供容器技术实现的 Docker daemon 及终端控制 Docker CLI 的应用程序。Docker Engine 主要用于 Docker 容器技术的实现，包括镜像构建和转移、容器启动和停止、网络控制等相关支持。

Docker Engine 为我们在不同的机器中实现了快速部署和搭建程序环境的方法，但提高效率只作用在了孤立的主机上，对于集群环境下的大规模部署，需要其他的方案来解决。为了辅助 Docker 在集群部署中体现出的高效性，Docker 官方也提供了多个与 Docker 相关的工具软件来辅助使用 Docker Engine，如 Docker Compose、Docker Machine、Docker Swarm 等。

在本章中，我们就介绍 Docker Compose、Docker Machine、Docker Swarm 这三个能够帮助我们更好地使用 Docker 的工具软件，看看它们是如何在各自的领域补足 Docker 功能上的缺失，又是如何提升工作效率的。

### 15.1 Docker Compose

Docker Compose 是管理和控制由多个容器所组成的服务体系的得力工具，通过它可以更好、更快地启动和停止大型项目，也能够更清晰地管理大型项目所包含的容器与容器之间的关系。

### 15.1.1 Docker Compose 简介

任何一个相对完整的应用服务都不可能以单一的程序来完成支持,使用 Docker 部署的服务更是如此,我们更推崇将大型服务拆分成较小的微服务,并将它们分别部署在不同的容器中。也就是说,完整的服务往往是由数个、数十个,甚至上百个不同的容器所提供的服务组成的。

对于集群化来说,我们首先要解决的就是迁移的问题。在之前的章节里,我们谈到了利用导入导出的方式来迁移镜像或容器。不过这种方式太过烦琐,不推荐使用。之后我们谈到了 Dockerfile,它小巧精干,也能构造完全镜像的一致体验,非常适用于对镜像的迁移。但是在组成服务的容器过多时,利用 Dockerfile 进行迁移仍然是一个烦琐的过程。因为 Dockerfile 只记录镜像的构建过程,并不记录容器的启动过程,而在实际迁移的过程中,我们还需要记住容器的启动参数等内容。若只有一两个容器,可以采取记住和输出这些参数的方式,但是若容器数量庞大,或者需要部署的机器众多时,这种方式就不可取了。

Docker Compose 就是用来简化多容器部署和迁移的过程的。Docker Compose 是由 Python 编写的程序,最初源于一个类似的开源项目 Fig,其定位就是定义和运行组合应用程序。

与 Dockerfile 类似,Docker Compose 也有一套独特的配置文件系统。不过,Dockerfile 定义的是镜像构建的过程,而 Docker Compose 配置文件定义的是容器的启动方式和配置,以及不同容器间的依赖关系。Docker Compose 程序正是根据我们在配置文件中的定义,来完成容器的启动过程的。

在开发、测试和持续集成中,Docker Compose 都是部署 Docker 容器的利器。因为我们能够很明确地通过配置文件,将所有组成服务的容器的配置,以及它们之间的依赖关系清晰地表示出来。并且通过一条简单的命令,就能完成整个项目的启动、停止、重启等操作。Docker Compose 会把我们给出的这条指令,结合 Docker Compose 项目的配置文件,转换成对应的 Docker API 的操作,直接体现到 Docker Daemon 中,这就代替我们完成了重复输入复杂指令的过程。

Docker Compose 有以下两个主要的组成定义。

- Service (服务)。服务代表的是运行同种应用程序的一个或多个相同容器的抽象定义,也是我们在 Docker Compose 中配置的主要对象。在每个 Docker Compose

的配置文件中，我们可以定义多个服务，并定义服务的配置，以及服务与服务之间的依赖关系。

- ❑ **Project（项目）。**项目代表的是由多个服务所组成的一个相对完整的业务单元，体现为一个 Docker Compose 配置文件。

Docker Compose 的配置默认存在于名为 `docker-compose.yml` 的文件中，我们也能够自定义文件名，并在使用命令时指定这个配置文件。与 `Dockerfile` 类似，使用 Docker Compose 的过程也与存放配置文件的目录息息相关。由于采用了环境目录这种形式，我们还能利用 Docker Compose 配置的特性，完成一些直接使用 Docker CLI 命令无法完成的事情，比如使用相对路径挂载宿主机目录或文件作为数据卷。

使用 Compose 时要先准备每个容器所基于的镜像，这些镜像可以通过远程仓库或本地仓库中已有的镜像来完成，也可以使用 `Dockerfile` 来定义。如果使用 `Dockerfile` 来定义 Compose 中的服务所基于的镜像，那么 Docker Compose 能够自动完成镜像的构建过程，也能够管理镜像的生命周期，以及在需要的时候重新构建镜像。

接着，要把服务（容器）的配置写入 `docker-compose.yml` 中。在配置文件中，我们可以定义容器所基于的镜像、挂载的目录、连接到的其他容器，以及绑定到宿主机上的端口等。我们可以定义特有的名称来区分不同的服务。

最后，通过 `docker-compose up` 命令，Docker Compose 就能完成“下载或构建镜像—项目基础环境—创建和启动容器”的整个过程。

如图 15-1 所示，使用 `docker-compose up` 命令就可以完成对整个由 Docker Compose 所搭建的项目的启动。

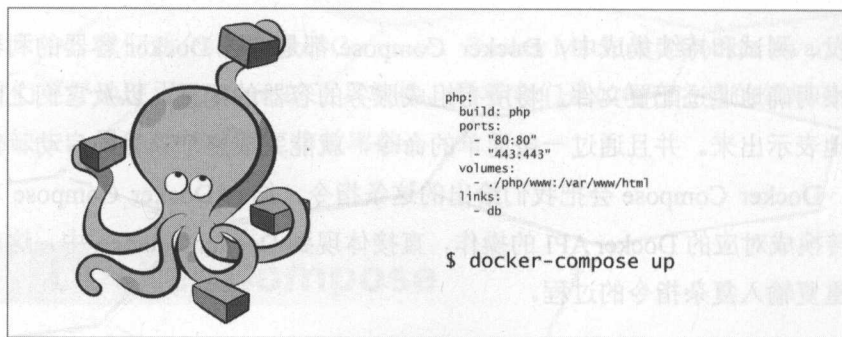


图 15-1 启动项目

Docker Compose 能够管理整个项目的生命周期，包括项目中所有服务容器的启动、停止、重启，甚至重建，也包括查看这些容器的运行状态及日志，还包括在这些容器中



执行命令。对于集群部署来说，Docker Compose 的作用是不可估量的，它可以真正完成在某一台宿主机上的一条指令部署。

### 15.1.2 安装 Docker Compose

目前，Docker Compose 的代码托管在 GitHub 上，其发布和下载也都在 GitHub 上进行。我们可以在 <https://github.com/docker/compose> 这个 GitHub 页面中，找到 Docker Compose 的代码，以及最新的发行版本和下载链接。我们也可以在终端或命令程序中输入下面的命令，来获得 Docker Compose：

```
$ sudo curl -L https://github.com/docker/compose/releases/download/1.8.0/docker-compose-`uname -s`-`uname -m` > /usr/local/bin/docker-compose
$ sudo chmod +x /usr/local/bin/docker-compose
```

我们可以根据需要修改这条命令中的路径。其中，1.8.0 代表 Docker Compose 的版本，我们可以根据需要下载对应的版本。

对于使用 Windows 和 Mac 的开发者，也就是安装了 Docker for Windows 或 Docker for Mac 的开发者来说，Docker Compose 已经附带在这两个软件之中。如图 15-2 所示，我们可以直接在 Windows 的 PowerShell 或 Mac OS 的 Terminal 中通过 docker-compose 命令来操作 Docker Compose。

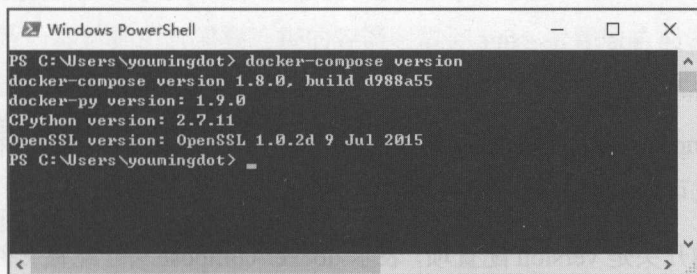


图 15-2 操作 Docker Compose

我们可以通过 docker-compose version 命令查看 Docker Compose 和其相关依赖软件库的版本；通过 Docker Compose 在 GitHub 页面更新 Docker Compose 程序；通过下载 Docker Compose 的命令，直接更新覆盖已有的版本。Docker Compose 的最新版本会随着 Docker 的更新一同更新到安装 Docker for Windows 或 Docker for Mac 的机器中，所以保持这两个软件的更新是更简单的方式。

### 15.1.3 Docker Compose 配置文件

使用 Docker Compose 最关键的步骤就是对 docker-compose.yml 进行编写。那么 Docker Compose 具体有哪些配置规则呢？下面通过运行组成 WordPress 的 docker-compose.yml 来展示常用的 Docker Compose 配置参数。

```
version: '2'
services:
  db:
    image: mysql:5.7
    volumes:
      - "./.data/db:/var/lib/mysql"
    environment:
      MYSQL_ROOT_PASSWORD: wordpress
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD: wordpress

  wordpress:
    build: ./wordpress
    links:
      - db
    ports:
      - "8000:80"
    environment:
      WORDPRESS_DB_HOST: db:3306
      WORDPRESS_DB_PASSWORD: wordpress
```

Docker Compose 的配置采用 YAML 格式，这是一种用于可读性的表征配置格式，对此不熟悉的朋友可以从网上找到相关的格式说明。

配置文件的开头是 version 配置项，表示 docker-compose.yml 配置文件使用的解析规则格式。目前，Docker Compose 所支持的解析格式有两个版本，支持定义配置较多且更规范的版本是 2，即这里采用的版本格式。

接下来是 services 项，表述的是所有组成项目的服务容器及它们的配置。这项是格式版本 2 中特有的，版本 1 中所有的服务都以配置项的顶级呈现。在所有容器的配置中，配置的键名就是我们为容器定义的名称，不过这个名称是用在 Docker Compose 里的，在实际生成容器时，Compose 会在容器名称前加上项目的名称，在之后加上项目的序号。在 WordPress 里有两个组成容器，一个是项目所使用的 MySQL 数据库容器，一个是运

行 WordPress 程序的容器。我们为 MySQL 容器命名为 db，为 WordPress 容器命名为 wordpress。

在单个服务的配置项中，我们可以看到 image 或 build 这两个配置项，它们都代表容器所基于的基础镜像。其中，image 代表从镜像仓库中查找对应的镜像，其值表示镜像的仓库名。build 表示通过指定的 Dockerfile 来构建镜像并使用，其值需提供 Dockerfile 构建的环境目录。

我们使用 volumes 配置项来挂载数据卷完成。配置值的书写方式与在 docker CLI 中使用的 -v 参数基本一致，不同之处在于，我们可以使用相对于 docker-compose.yml 文件的路径挂载宿主机中的目录或文件。而在 Docker CLI 中，因为数据卷名称与相对路径冲突，所以是无法在 -v 参数中使用相对路径的。

容器中的环境变量可以通过 environment 参数来指定，只要把环境变量的名称和值以键值对的形式书写到 docker-compose.yml 文件中即可。

要映射主机的端口可以通过 ports 配置项来实现。指定的方式与 Docker CLI 中的 -p 参数类似，需要映射多个端口时，可以采用 YAML 的列表形式传入。需要特别注意的是，因为 YAML 格式解析的关系，在书写端口映射时是可以不使用双引号的。但是当使用的端口号小于 60 时，直接在 YAML 中使用数字会被理解为时间，所以最好还是统一使用双引号包裹端口映射的配置，避免出现问题。

容器间的连接可以使用 links 配置项来实现。其使用方法非常简单，和在 Docker CLI 中使用 --link 参数一样，把所有需要连接的容器名放到 links 配置项的列表值中即可。

## 15.1.4 常用的 Docker Compose 命令

### 1. 构建项目镜像

通过 docker-compose build 命令，可以对 Docker Compose 项目中需要且由 Dockerfile 提供的镜像进行构建。构建的镜像都是在 Docker Compose 配置文件中通过 build 字段定义的。

```
$ sudo docker-compose build
```

通过 --pull 参数，可以在构建的过程中，总是从远程仓库中拉取依赖的基础镜像的最新版本。

通过 --force-rm 参数，Docker Compose 会在构建镜像之前移除所有基于原有镜像的容器。



通过`--no-cache` 参数，可以让构建镜像的过程不使用构建缓存。

## 2. 创建项目容器

使用 `docker-compose create` 命令，可以让 Docker Compose 根据项目配置文件创建相应的容器。在创建容器的过程中，Docker Compose 会自动解决容器之间的依赖，完成对数据卷、网络等模块的配置，并且可以进行容器连接和端口映射等工作。

```
$ sudo docker-compose create
```

通过`--force-create` 参数，可以让 Docker Compose 重新创建已经存在于 Docker 中的容器。

通过`--build` 参数，可以让 Docker Compose 构建容器所需的镜像，即使这些镜像已经存在。

## 3. 运行项目

使用 `docker-compose start` 命令，可以启动 Docker Compose 项目，也就是启动 Docker Compose 定义的所有容器。

```
$ sudo docker-compose start
```

如果只启动某几个特定的服务，可以在命令之后加上需要启动的服务列表。

```
$ sudo docker-compose start nginx php mysql
```

## 4. 组合运行

使用 `docker-compose up` 命令，可以让 Docker Compose 完成 Docker Compose 项目的镜像构建、容器创建、容器启动和连接容器到容器。可以说，组合运行能够通过一条命令完成整个项目的运行，`docker-compose up` 命令是最常用的启动 Compose 项目的命令。

```
$ sudo docker-compose up
```

通过`-d` 参数，可以让 Docker Compose 项目以后台方式运行。在默认情况下，`docker-compose up` 命令会在前台运行项目，所有的容器输出会及时显示到终端中。

通过`--force-create` 参数，可以让 Docker Compose 重新创建已经存在于 Docker 中的容器。

通过`--build` 参数，可以让 Docker Compose 构建容器所需的镜像，即使这些镜像已经存在。

## 5. 停止项目

使用 `docker-compose stop` 命令，可以停止 Compose 项目的运行。

```
$ sudo docker-compose stop
```

通过 `--timeout` 参数，可以设置需要停止的容器在停止过程中的等待超时时间。

## 6. 删除项目

使用 `docker-compose rm` 命令，可以删除 Docker Compose 项目所创建的容器。

```
$ sudo docker-compose rm
```

通过 `--force` 参数，可以进行强制删除，其效果与 `docker rm` 命令中的 `-f`、`--force` 参数是基本一致的。

通过 `-v` 参数，可以同时判断容器所使用的数据卷是否还存在引用，没有其他容器引用的数据卷会一并进行删除。

通过 `--all` 参数，可以让通过 `docker-compose run` 命令启动的容器也被删除。

## 7. 组合结束

通过 `docker-compose down` 命令，可以让 Docker Compose 项目中所有通过 `docker-compose up` 命令启动的容器停止，对所创建的镜像及网络进行删除。这个命令与 `docker-compose up` 命令成对使用，是完成项目运行和停止最快的操作方案。

```
$ sudo docker-compose down
```

通过 `--remove-orphans` 参数，可以删除未被 Docker Compose 配置文件定义的容器，这对于改变 Docker Compose 配置后进行的 `docker-compose down` 命令非常有用。

## 8. 获得服务日志

通过 `docker-compose logs` 命令，可以得到容器输出的内容。由于 Docker Compose 会自动管理项目中的容器，而其管理的容器命名与项目中配置的可能存在差异，所以最好选择 `docker-compose logs` 命令来查看容器的输出。

```
$ sudo docker-compose logs <service>
```

`docker-compose logs` 命令接收 Docker Compose 配置中的服务名称，如果一个服务生成了多个容器，则所有的容器输出也会合并显示。同时，也能够给出多个服务名称，而且这些服务的输出也会合并显示出来。

通过`--follow` 参数，可以让 Docker Compose 进行持续的输出。

通过`--timestamps` 参数，可以显示出每条输出的时间信息。

通过`--tail` 参数，可以截断输出的最后几行进行显示，截断的方式细腻化到容器，如果给出此参数并且存在多个容器合并输出，则每个容器的最后几行都会被显示出来。

## 15.2 Docker Machine

Docker Machine 可以在更多非 Linux 核心的系统中搭建 Docker 环境，它实现了虚拟机容器虚拟化的组合搭配。在很多场合下，特别是集群部署在非 Linux 核心的系统主机中时，它能够为我们带来很大的帮助。

### 15.2.1 Docker Machine 简介

Docker 已经把容器技术发挥到了极致，通过 Docker 和其生态圈中的相关软件，可以非常轻松地地为程序搭建最适合的环境，并且把这个环境轻松地复制到其他机器上。但想要使用这些便捷的优势，有一个最基础的条件——机器上安装了 Docker。

Docker 实现容器化，基于的是 Linux Container 及其相关技术。而 LXC 技术又是 Linux Kernel 所特有的实现，在其他非 Linux 系统中是没有这样的支持的。那么要在非 Linux 系统中安装和使用 Docker，就必须通过虚拟机模拟出一套 Linux 系统出来。在之前的章节中，我们讲过在 Windows 和 Mac OS 系统中安装和使用 Docker，但在这两个系统中安装和配置的方法是不一样的。那么在实际生产中，特别是在分布式部署中，就要使用不同系统的主机，Windows 和 Mac OS 只是其中的一部分，AWS、Digital Ocean 等云计算提供商的系统方案也有所不同。所以，对于集群部署来说，提高效率不仅体现在提高对服务程序的部署上，还体现在对 Docker 的安装、更新和配置上。

Docker 可以让容器中的应用程序处于与外界隔离的运行环境中，这让应用程序可以随着容器提供的环境数据迁移到其他主机的 Docker 中。但是，要把 Docker 程序本身部署到软硬件环境不同的主机中去，却不能采用把 Docker 放在 Docker 容器里来进行迁移的方式。好在 Docker 为我们提供了 Docker Machine 这个程序，即简化对集群环境下运行在不同主机上的 Docker 进行管控的操作。



Docker 的迭代速度很快，要想保证运行在 Docker 容器中的应用程序得到一致的运行体验，首先就要保证运行这些容器的 Docker 在功能配置上是一致的。如图 15-3 所示，以前，如果需要将 Docker 部署到不同的机器上，特别是主机环境本身就不同的机器上，就需要单独进行操作。也就是说，需要在每台机器上安装和调试 Docker，才能确保不同主机中的 Docker 容器有运行一致的配置。

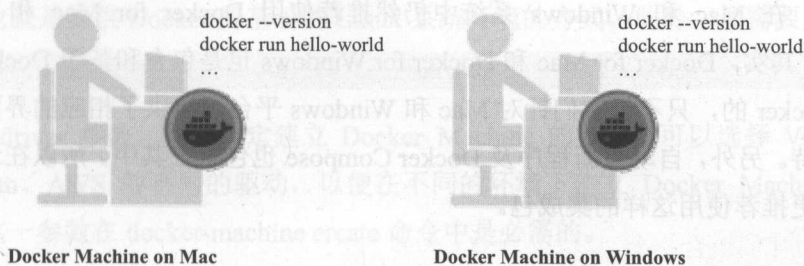


图 15-3 在不同的机器上部署 Docker

而有了 Docker Machine 之后，我们就可以通过它统一管理运行在不同机器上的 Docker，如图 15-4 所示。

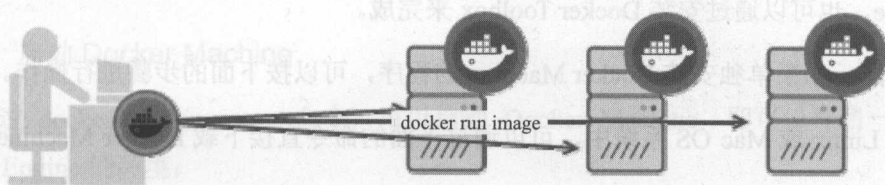


图 15-4 利用 Docker Machine 统一管理运行 Docker

我们可以把 Docker Machine 理解为 docker daemon 与 Docker CLI 分离的程序。Docker Machine 的客户端通过连接主机端来操作 Docker，这种连接方式可以通过网络进行。如图 15-5 所示，可以远程操作 Docker 所在的宿主机，而不用再登录到这台机器上。

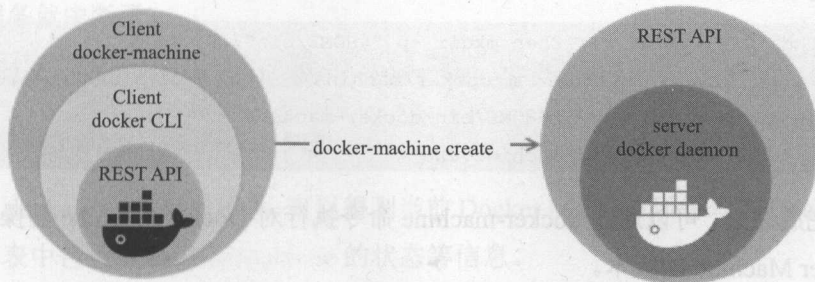


图 15-5 远程操作 Docker 所在的宿主机

Docker Machine 的另外一个特点,就是它提供了在虚拟机上搭建 Docker 的整个流程。因为 Docker 无法在非 Linux 内核中运行,所以要在这些环境下运行 Docker,要先通过虚拟机搭建一套 Linux 系统。而只要通过 Docker Machine 指定用于搭建 Linux 虚拟机的驱动, Docker Machine 就能自动完成从创建虚拟机到 Docker 的安装与配置的操作。在 Windows、Mac OS 乃至一些云平台的主机中,使用 Docker Machine 的作用是显而易见的。

另外,在 Mac 和 Windows 系统中仍然推荐使用 Docker for Mac 和 Docker for Windows。其实, Docker for Mac 和 Docker for Windows 也是包含和基于 Docker Machine 来实现 Docker 的,只不过它们针对 Mac 和 Windows 平台,提供了相应的界面化操作和配置的支持。另外,自动更新程序及 Docker Compose 也包含在其中。所以在这两个系统平台下,更推荐使用这样的集成包。

## 15.2.2 安装 Docker Machine

要使用 Docker Machine,首先就要安装它。在 Mac OS 和 Windows 系统中,如果不想通过 Docker for Mac 或 Docker for Windows 来使用 Docker,而仅仅只是要安装 Docker Machine,也可以通过安装 Docker Toolbox 来完成。

当然,如果单独安装 Docker Machine 的程序,可以按下面的步骤进行操作。

在 Linux 或 Mac OS 系统中,可以通过下面的命令直接下载 Docker Machine 程序并使用。

```
$ sudo curl -L https://github.com/docker/machine/releases/download/v0.8.0/docker-machine-`uname -s`-`uname -m` > /usr/local/bin/docker-machine && \
$ sudo chmod +x /usr/local/bin/docker-machine
```

在 Windows 系统中也可以通过 Git Bash 或在相似的命令行工具中输入下面的命令来下载和使用 Docker Machine。

```
$ if [[ ! -d "$HOME/bin" ]]; then mkdir -p "$HOME/bin"; fi && \
$ curl -L https://github.com/docker/machine/releases/download/v0.8.0/docker-machine-Windows-x86_64.exe > "$HOME/bin/docker-machine.exe" && \
$ chmod +x "$HOME/bin/docker-machine.exe"
```

安装完成之后,可以通过 docker-machine 命令执行对 Docker Machine 的操作。例如,查看 Docker Machine 的版本。

```
$ docker-machine version
docker-machine version 0.8.0, build b85aac1
```

## 15.2.3 Docker Machine 常见命令

### 1. 创建 Docker Machine

通过 `docker-machine create` 命令，可以在已经安装了 Docker Machine 的机器上创建一个运行 Docker 的环境。创建 Docker Machine 的时候，必须给出 Docker Machine 基于的驱动，也就是提供 Docker 所基于的 Linux 系统环境的方式。此外，还需要为其命名。

```
$ sudo docker-machine create --driver virtualbox machinewithvbox
```

通过 `--driver` 参数，能够指定建立 Docker Machine 的方式，可以选择 VirtualBox、DigitalOcean、AWS 等不同的驱动，以便在不同的环境下建立 Docker Machine 并使用 Docker。这一参数在 `docker-machine create` 命令中是必需的。

通过 `--engine-storage-driver` 参数，可以给出 Docker Machine 中 Docker Engine 使用的存储驱动。

通过 `--engine-registry-mirror` 参数，可以指定 Docker Machine 中 Docker Engine 远程镜像仓库的镜像地址。

### 2. 启动 Docker Machine

通过 `docker-machine start` 命令，可以启动 Docker Machine，即可以开启一个运行 Docker Engine 的环境。

```
$ sudo docker-machine start machinewithvbox
```

### 3. 停止 Docker Machine

通过 `docker-machine stop` 命令，可以停止 Docker Machine 的运行。这就意味着，如果有 Docker daemon 程序正运行在这个 Docker Machine 中，那么它也会随之停止，Docker 服务就中断了。

```
$ sudo docker-machine stop machinewithvbox
```

### 4. 获得 Docker Machine 列表

通过 `docker-machine ls` 命令，可以得到当前 Docker Machine 所管理的 Docker Machine 列表，列表中包括了 Docker Machine 的状态等信息。

```
$ sudo docker-machine ls
```

通过 `--timeout` 参数，能够限制 Docker Machine 检查 Docker Machine 状态的超时时间。



通过`--filter` 参数，能够对列表返回结果进行过滤。

## 5. 查看 Docker Machine 的状态

通过 `docker-machine status` 命令，能够获得指定 Docker Machine 的运行状态。

```
$ sudo docker-machine status machinewithvbox
```

## 6. 更新 Docker Machine

通过 `docker-machine upgrade` 命令，能够对指定的 Docker Machine 进行更新。Docker Machine 会对 Machine 从不同的维度进行更新，包括对运行在 Machine 中的 Docker 进行更新。对于不同的 Machine 实现形式，Docker Machine 会采用不同的更新方式，比如通过 `boot2docker` 实现 Linux 环境的 Machine，在进行更新时也会对 `boot2docker` 的镜像进行更新。

```
$ sudo docker-machine upgrade machinewithvbox
```

# 15.3 Docker Swarm

如果要在由多台运行 Docker 主机组成的集群上部署相同或相似的容器或项目，或要对这些 Docker 服务同时进行配置，就不得不借助一些工具，在这些工具中，首屈一指的当属 Docker 官方提供的 Docker Swarm。

## 15.3.1 Docker Swarm 简介

Docker Swarm 是专门用于 Docker 集群管理的工具，能够将对多个分布在不同地域甚至不同时间中的 Docker 服务的操作，转化在一台主机上使用 Docker Swarm 进行。也就是说，Docker Swarm 能够把给出的 Docker 命令，下放到所有关联 Docker Swarm 上的 Docker 服务中。

在 Docker Swarm 中执行命令的形式与在 Docker CLI 中执行命令的形式几乎一致，因此当我们使用 Docker Swarm 时，可以当成使用 Docker CLI 对单一的 Docker 服务的操作。更可贵的是，因为 Docker Swarm 有与 Docker 相似的操作，所以它可以兼容大部分 Docker 工具。我们可以把这些工具作用在 Docker Swarm 上，然后所有的操作被分别推送到集群中的主机上。

### 15.3.2 Docker Swarm 结构

如图 15-6 所示，在 Docker Swarm 的架构中，主要有三种角色：Manager、Cluster 和 Node。

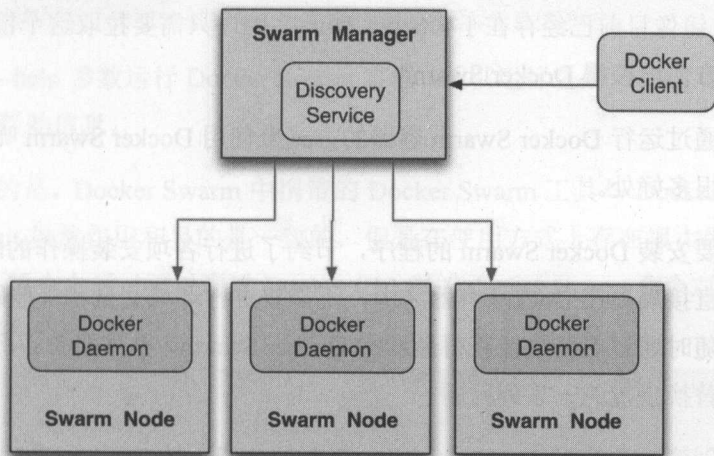


图 15-6 Docker Swarm 的体系结构

其中，Swarm Node 的作用就是包裹 Docker Daemon，让 Docker Swarm 在对 Docker Daemon 进行操作时能够达到一致的体验。Swarm Cluster 表示由多个 Swarm Node 组成的集群，也就是使用 Docker Swarm 主要操作的对象。Swarm Manager 就是实现对 Swarm Cluster 进行操作的终端管理程序。

Docker Swarm 可以实现一个完全透明的中间件，我们能够通过 Docker CLI 对 Docker Swarm 中的 Swarm Manager 进行操作，这些操作与使用 Docker CLI 对 Docker Daemon 进行操作的体验是一致的。而 Swarm Manager 也能够将操作指令传达给 Swarm Node，并由 Swarm Node 对 Docker Daemon 进行相应的操作。

在整个操作过程中，Docker CLI 的操作与平时进行的操作是一致的，而 Docker Daemon 收到指令的方式与通常的接收指令的方式是一致的。

### 15.3.3 使用 Docker Swarm

由于 Docker 本身就是常用于分布式及集群化的部署工具，所以作为 Docker 集群管理的 Docker Swarm 也是常见和常用的 Docker 辅助工具之一。在 Docker 1.12 版本之后，Docker Swarm 已经被集成为 swarmkit 子模块，并随 Docker 一同安装。所以在 Docker 1.12

版本中，我们能够直接通过命令创建和管理 Docker Swarm，不需要再依赖其他独立的 Docker Swarm 工具程序。

而在 Docker 1.12 之前的版本中，我们也可以通过很多方式来获得 Docker Swarm。最简单的方式莫过于通过 Docker Swarm 的 Docker 镜像获得和使用 Docker Swarm。Docker Swarm 镜像目前已经存在于 Docker Hub 之中，只需要拉取这个镜像并运行容器，就可以很方便地获得 Docker Swarm。

为什么要通过运行 Docker Swarm 容器的方式来使用 Docker Swarm 呢？因为使用这种方式会带来很多好处。

- ❑ 不需要安装 Docker Swarm 的程序，节约了进行各项安装操作的时间。
- ❑ 程序直接隔离在 Docker 容器之中，不需要进行环境变量和程序路径等的配置。
- ❑ 能够随时通过更新镜像获得最新的 Docker Swarm，不需要关心旧程序删除和新程序替换引发的一系列问题。

下面直接创建一个基于 Docker Swarm 镜像的容器，让 Docker 自动完成对 Docker Swarm 镜像的下载。

```
$ sudo docker run swarm --help
Unable to find image 'swarm:latest' locally
latest: Pulling from library/swarm
220609e0bc51: Pull complete
b54bf338fe2f: Pull complete
d53aac5750d5: Pull complete
Digest: sha256:c9e1b4d4e399946c0542accf30f9a73500d6b0b075e152ed1c792214d3509d70
Status: Downloaded newer image for swarm:latest
Usage: swarm [OPTIONS] COMMAND [arg...]

A Docker-native clustering system

Version: 1.2.5 (27968ed)

Options:
  --debug                debug mode [$DEBUG]
  --log-level, -l "info" Log level (options: debug, info, warn, error, fatal, panic)
  --experimental         enable experimental features
  --help, -h            show help
  --version, -v          print the version

Commands:
```



```

create, c Create a cluster
list, l List nodes in a cluster
manage, m Manage a docker cluster
join, j Join a docker cluster
help Shows a list of commands or help for one command

```

Run 'swarm COMMAND --help' for more information on a command.

这里使用--help 参数运行 Docker Swarm 容器，所以运行之后会在终端中看到 Docker Swarm 的使用帮助信息。

需要注意的是，Docker Swarm 中携带的 Docker Swarm 工具与 Docker 1.12 版本之后内置的 swarmkit 虽然作用和目的是一致的，但是在使用方式上存在很大的区别。例如，在 Docker 1.12 版本之后，可以通过 Docker CLI 中的 docker swarm 命令对 swarmkit 所提供的集群管理进行操作。

```
$ sudo docker swarm --help
```

```
Usage: docker swarm COMMAND
```

```
Manage Docker Swarm
```

```
Options:
```

```
--help Print usage
```

```
Commands:
```

```

init      Initialize a swarm
join      Join a swarm as a node and/or manager
join-token Manage join tokens
update    Update the swarm
leave     Leave a swarm

```

Run 'docker swarm COMMAND --help' for more information on a command.

## 15.3.4 Docker Swarm 常见命令

### 1. 创建 Docker Swarm

使用 Docker Swarm 需要先拥有一个 Swarm，并建立一个可以管理所有 Node 的 Swarm Manager。

Docker 1.12 版本之后的 Docker, 可以通过 Docker CLI 中的 `docker swarm init` 命令初始化 Swarm Manager 模块。

```
$ sudo docker swarm init
Swarm initialized: current node (awlgu0xrfpbg7m7a377cne0e3) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join \
        --token
        SWMTKN-1-2cionv3ujfu4t9rmvzq7plnfoefvpxxp4fni8ly4apcb3zymaz-28wap3eo2tzzm9iwk76do
        bh89 \
        10.0.75.2:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the
instructions.
```

初始化的操作分为两步, 首先要创建一个新的 Docker Swarm, 之后再将当前的 Docker Engine 加入到 Docker Swarm 中并作为 Swarm Manager。初始化完成后, 当前的 Docker Engine 就变成了一个新 Docker Swarm 的管理节点, 承担着 Swarm Manager 的角色。每个 Docker Swarm 都会拥有一个特有的 Token, 用于 Node 的加入。我们创建 Swarm 时 Token 会自动生成, 需要记录下这个 Token, 以便在之后的操作中使用。

如果是通过 Docker Swarm 容器使用 Docker Swarm 的, 那么创建 Swarm Manager 就要单独完成上面提到的两个步骤。首先要创建一个新的 Swarm。

```
$ sudo docker run --rm swarm create
86222732d62b6868d441d430aee4f055
```

创建新的 Swarm 后可以得到一串字符, 这就是 Swarm 的 Token。我们要记录下这个 Token, 以便之后使用。

然后需要创建一个 Swarm Manager 并加入到刚才创建的 Swarm 中。

```
$ sudo docker run swarm manage token://86222732d62b6868d441d430aee4f055
```

这个命令就使用到了刚才记录的 Swarm 的 Token。

## 2. 加入 Docker Swarm

要让 Docker Engine 作为 Docker Swarm 的 Node 加入到 Docker Swarm 中, 在 Docker 1.12 版本以上的 Docker Engine 里, 可以通过 Docker CLI 中的 `docker swarm join` 命令来实现。

```
$ sudo docker swarm join --token SWMTKN-1-2cionv3ujfu4t9rmvzq7plnfoefvpxxp4fni8ly
4apcb3zymaz-28wap3eo2tzzm9iwk76dobh89 10.0.75.2:2377
```

如果我们通过 Docker Swarm 镜像来实现 Docker Swarm 的功能，则可以通过运行 Docker Swarm 容器的 join 方法来加入到 Swarm 中。

```
$ sudo docker run -d swarm join --addr=node1:2376 token://86222732d62b6868d441d
430aee4f055
```

### 3. 离开 Docker Swarm

如果想要让 Node 离开 Docker Swarm，可以通过在 Swarm Node 所在的 Docker Engine 中运行 `docker swarm leave` 命令来实现。

```
$ sudo docker swarm leave
```

如果是通过 Docker Swarm 镜像完成 Docker Swarm 搭建的，只需要关闭对应节点的 Docker Swarm 容器即可。

## 15.4 本章小结

在本章中，我们对 Docker 中比较常用的三个辅助和管理工具进行了介绍。其中，有能够帮助我们快速部署、管理由多个容器组成项目的 Docker Compose，也有能够帮助我们在不同平台中安装和统一管理 Docker 程序的 Docker Machine，还有能够帮助我们在管理集群中使用 Docker 部署的 Docker Swarm。

通过本章的学习，我们对这三个最常用的 Docker 管理工具的基本知识有了了解，也对它们各自的功能和用途进行了说明。这三个工具分别从不同的角度，帮助我们将 Docker 从单机的容器隔离与虚拟化，转向在大规模集群中部署应用。掌握好 Docker Compose、Docker Machine 和 Docker Swarm 的使用方法，是熟练使用 Docker 的基础。



# 第 16 章

## Docker 的技术架构

Docker 的容器技术来源于 Linux Container 的支持。那么，Docker 使用到了 Linux Container 中的哪些技术？在此基础之上，又进行了哪些延伸性的开发呢？在之前的章节中，我们零零散散地说到过 Docker 几项比较重要的核心技术，但都没有进行深入讲解。本章专门针对命名空间（Namespaces）、控制组（CGroups）、联合文件系统（Union File System）及 Docker Engine 程序的架构进行介绍。

### 16.1 命名空间

在操作系统中，网络配置、进程、用户、IPC（进程间调用）等信息和操作，都是可以被所有程序查看到的。如果需要通过容器虚拟化，除了需要控制内存、CPU、文件系统及操作系统中的其他资源，还需要隔离进程之间能够共享的信息。Namespaces 正是用于实现进程间信息隔离的。

#### 16.1.1 关于 Linux 命名空间

因为涉及的大多数信息是由操作系统管理的，所以实现进程间信息的独立与隔离需要在操作系统内核层面进行实现。如图 16-1 所示，Linux 中的 Namespaces 是 Linux 为了实现容器虚拟化而引入到内核中的一种内核级别的环境隔离方法，在 Linux Kernel 迭代

的过程中逐渐增加到了 Linux Kernel 中，并由最开始的挂载隔离逐渐发展成由多种命名空间组成的体系。

进程 1-1	进程 1-2	进程 2-1	进程 2-2	进程 3-1	进程 3-2
Namespaces 1		Namespaces 2		Namespaces 3	
Linux Kernel					
硬件					

图 16-1 Linux Namespaces

Linux Kernel 所提供的 Namespaces 主要分为以下几种。

- ❑ Mount Namespaces: 挂载命名空间，用于隔离挂载目录。
- ❑ UTS Namespaces: UTS 命名空间，用于隔离主机及域等信息。
- ❑ IPC Namespaces: IPC 命名空间，用于隔离进程间的调用。
- ❑ PID Namespaces: 进程命名空间，用于隔离进程的运行信息。
- ❑ Network Namespace: 网络命名空间，用于隔离网络配置和访问。
- ❑ User Namespace: 用户命名空间，用于隔离用户和用户组信息。

## 16.1.2 命名空间的系统调用

系统调用 (System Call) 是指运行在操作系统中的程序，调用操作系统所提供的应用程序编程接口 (Application Programming Interface)，来实现对操作系统下达指令和获取信息的过程。操作系统与 CPU 有类似的机制，分为核心态和用户态，这两种状态在保障程序正常运行的同时，限制程序直接进行一些比较敏感的操作。通常来说，程序都运行在用户态下，而当程序需要进行影响操作系统运行或者其他受到保护的操作时，都需要通过操作系统所提供的接口来进行。也就是说，对于操作系统敏感部分的操作，使用系统调用的方式去实现，在一定程度上保障了程序进行这些操作时的安全性。

使用 Namespaces 时，在 Linux Kernel 提供的接口方法中，主要囊括了三种系统调用的方法。

- ❑ setns: 把进程加入指定的 Namespaces 中。
- ❑ unshare: 使进程脱离指定的 Namespaces。
- ❑ clone: 实现线程的系统调用，用来创建一个新的进程，并通过设计上述参数达到隔离。

如果想通过 Linux Kernel 的 Namespaces 实现程序隔离，就离不开这三个方法。而

Docker Daemon 正是通过这种方式，将运行在不同容器中的进程，分配到了不同的内核命名空间中。

## 16.1.3 命名空间的分类

### 1. 挂载命名空间

Linux 的文件系统是 Linux Kernel 中非常重要的一部分，它通过统一接口的方式让 Linux 能够很容易地兼容各种真实的文件系统。在 Linux 中，文件目录树是基于挂载（Mount）操作实现的，也就是所有文件系统的目录来源于实际的数据存储器。

运行在 Linux 中的进程，能够通过指定的方法读取目录树，也能够修改挂载点。这一点看似很平常，但是如果要进行进程隔离，问题就会显现出来。

如果隔离在某个 Namespaces 中的程序，修改了挂载目录或其他影响目录树结构的信息，则在另外一个 Namespaces 中运行的程序，也能够察觉这样的修改。那么这个封闭在 Namespaces 中的程序对挂载信息的修改，无形中影响到了其他 Namespaces 中程序的运行，这显然是隔离不到位的结果。

所以要进行程序隔离，首先是把程序所使用的挂载目录树进行隔离，让不同的 Namespaces 拥有独立的挂载结构。而程序对挂载信息的修改，也不会影响其他 Namespaces 中程序的运行。挂载命名空间（Mount Namespaces）就是用来实现这个目的的。挂载命名空间是在 Linux Kernel 2.4.19 版本中加入的，是最早加入 Linux 命名空间中的技术实现。

### 2. UTS 命名空间

通过 UTS 命名空间，可以为不同的 Namespaces 设置不同的主机名与网络域，能够简单地将程序隔离到一个独立的网络空间。

在 Docker 中，可以在 `docker inspect` 命令所给出的容器信息中，找到容器的主机名。

```
$ sudo docker ps -a
CONTAINER ID IMAGE      COMMAND      CREATED      STATUS      PORTS
NAMES
5b0aa85e08b0      centos      "/bin/bash"  6 days ago   Exited (0)  3 seconds ago
gloomy_spence
$ sudo docker inspect -f "{{.Config.Hostname}}" gloomy_spence
5b0aa85e08b0
```



在默认情况下，Docker 使用了容器 ID 的前 12 个字符，作为容器隔离环境中的主机名。

### 3. IPC 命名空间

IPC (Inter-Process Communication, 进程间通信) 是 Linux 系统中最常见的进程直接互相交换信息的方式。采用 Namespaces 隔离进程运行的环境，也要避免程序通过进程间通信来访问其他 Namespaces 中的进程，否则就达不到隔离的目的了。

在 Linux Namespaces 体系中，可以通过 IPC Namespaces 实现隔离进程间通信的目的。IPC Namespaces 为每一个 Namespaces 提供了独立的系统信号量、消息队列和共享内存等所有用于 IPC 的信息。对于需要进行进程间通信的程序，它只能与同一命名空间下的程序通信，而不在同一命名空间下的进程，则无法通过进程间通信进行信息交换。

### 4. 进程命名空间

进程是程序运行最直接的体现方式，所以要实现程序的隔离，将进程信息进行隔离是重中之重。在 Linux Namespaces 中，可以通过 PID Namespaces 来实现进程信息的隔离。

为了最大化地节约转换的损耗，每一个运行在 Namespaces 中的进程，其实就真实运行在 Linux 系统中，所以我们在 Linux 系统中可以看到运行在 Namespaces 中的所有进程。

```
$ sudo docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
f6fle4440294	nginx	"nginx"	7 weeks ago	Up 7 weeks
	0.0.0.0:80->80/tcp, 0.0.0.0:443->443/tcp	nginx		

```
$ sudo docker exec nginx ps aux
```

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
root	1	0.0	0.2	36968	4072	?	Ss	Aug12	0:00	nginx: master process nginx
nginx	5	0.0	0.3	39920	6232	?	S	Aug12	0:16	nginx: worker process
root	6	0.0	0.0	17492	1172	?	Rs	16:31	0:00	ps aux

```
$ sudo ps aux | grep nginx
```

root	4686	0.0	0.2	36968	4072	?	Ss	Aug12	0:00	nginx: master process nginx
nginx	4738	0.0	0.3	39920	6232	?	S	Aug12	0:16	nginx: worker process
root	5669	0.0	0.0	112652	964	pts/0	S+	16:31	0:00	grep --color=auto nginx

虽然我们从宿主机系统中找到了 Namespaces 中运行的进程，但是 Namespaces 中的进程 ID (PID) 与宿主机系统中的进程 ID 并不相同，这就得益于 PID Namespaces 实现的进程信息隔离。PID Namespaces 为命名空间设置一个独立的进程管理栈，其中就包括了独立的进程号管理。每一个运行在 Namespaces 中的进程，会分配到一个属于这个命名

空间的进程 ID。所以我们在命名空间内部和命名空间外部查看同一个进程时，得到的进程 PID 并不是统一的。

## 5. 网络命名空间

UTS 命名空间只做到了简单的网络隔离，没有完全隔离网络配置等信息，所以 Linux 又提供了网络命名空间 (Network Namespaces) 来实现网络的隔离。在之前谈及的 Docker 中，在对容器的网络访问及 Docker 的高级网络使用中，我们多次介绍了 Network Namespaces 的作用。对于如何使用 Network Namespaces 来造就容器中的网络隔离环境，以及如何打破 Network Namespaces 的隔离实现不同容器间的网络通信，我们也详细进行了讲解，所以这里就不再展开。

## 6. 用户命名空间

在 Linux 中，用户权限是控制程序进行安全访问最简单、最直接的方法。只要运行在 Linux 系统中的程序拥有足够的权限，它就能通过系统接口实现对操作系统中的用户进行增删和修改等操作。而有一部分程序是需要通过用户控制来实现需求的，所以在 Namespaces 中无法对这些操作进行一刀切，而是需要通过专门的用户隔离机制，防止运行在 Namespaces 中的程序不能直接操作宿主机系统中的用户，以避免影响其他 Namespaces 中运行的程序。用户命名空间 (User Namespaces) 就是用来实现这个目的的。

在每一个通过 User Namespaces 隔离的环境中都存在独立用户体系，包括独立的用户及用户组，甚至是根账号。运行在 Namespaces 中的程序不必担心自己对用户的操作会影响到宿主机系统，因为这些操作都不会直接运用到宿主机系统上。

# 16.2 控制组

在 Linux Kernel 所提供的容器技术实现中，除了能够隔离进程运行的 Namespaces，还有能够控制资源使用的 CGroups。虚拟化中很关键的一个功能，就是对计算机资源的汇总和再分配，而 CGroups 就是用来完成这个任务的。

## 16.2.1 关于 Linux 控制组

CGroups (Control Groups, 控制组群) 的作用是记录、限制、隔离进程所使用的 CPU、

内存、文件 IO 等计算机资源。CGroups 最早称为 Process Container，由 Google 的程序员在 2006 年提出，后来为了避免与容器的概念冲突，改名为 CGroups，并在 2.6.24 版本中加入 Linux Kernel。

CGroups 通过插入程序对计算机硬件资源调用的过程，实现了控制进程使用资源的目的。作为 Linux Container 的两大关键技术，Namespaces 主要提供了进程信息、用户、挂载目录、网络配置等软件资源的隔离，而 CGroups 则为 CPU、内存、文件 IO 等硬件资源提供了隔离。

控制资源是容器技术中非常关键且非常需要的，我们可以通过 Namespaces 来实现不同的容器中的进程互相不能进行访问。但是如果缺少 CGroups 对硬件资源的控制，就避免不了容器中的程序对资源进行挤占，影响其他容器中程序的运行。特别是对于 CPU、内存及文件 IO 这几种常见且程序运行不可或缺的计算机资源，我们更不能掉以轻心。所以，CGroups 既是 Linux Container 重要的组成部分，也是 Docker 基础结构中的重要环节。我们在 Docker 中能够对每个容器占用的 CPU、内存等资源进行精确控制，正是得益于 CGroups 的支持。

## 16.2.2 CGroups 的组成

从功能上来分析，CGroups 主要有以下几个功能。

- ❑ **Resource Limiting:** 资源限制。CGroups 能够为每一个控制组设置使用资源的上限，一旦控制组中分配的此类资源达到了限制，就不会再分配更多的资源给这个控制组。比如，当控制组中的内存已经达到设置的上限，则控制组中的程序无法再分配新的内存。
- ❑ **Prioritization:** 优先级分配。CGroups 可以为不同的控制组设置不同的优先级。优先级较高的控制组，可以优先得到 CGroups 为其分配的计算机资源。
- ❑ **Accounting:** 资源统计。对于每个控制组，CGroups 都能掌握它们对资源的使用情况，如 CPU 使用时长、内存消耗、句柄数等。
- ❑ **Control:** 进程控制。CGroups 还能对控制组中的进程进行挂起、恢复等操作。

早期的 CGroups 还实现了与 Namespaces 类似的进程信息隔离的功能，不过后来因为此功能与 Namespaces 所实现的功能有所冲突，也为了更清晰地划分 Namespaces 用于软件资源而 CGroups 用于硬件资源，所以 CGroups 的此项功能被从 Linux Kernel 的实现中移除。



CGroups 有以下几个重要的组成结构。

- ❑ Task: 任务。分配计算机资源就是为了执行特定的任务，而这些任务具体到计算机中就是运行中的进程。
- ❑ CGroup: 控制组。由任务组成的任务组称为控制组，在 Cgroups 中，控制组是分配和控制资源分配的单位。每个控制组中包含了多个子系统。
- ❑ Subsystem: 子系统，也就是资源调度的控制器。比如控制 CPU 可以通过 CPU 子系统，限制内存可以通过内存子系统。
- ❑ Hierarchy: 继承树，用于处理控制组和子系统的关系。它能将控制组制成树状关系，并让子系统依附到对应的控制组上。

在 CGroups 中，还有任务 (Task)、CGroup (任务组)、Subsystem (子系统)、Hierarchy (继承树) 等概念，如果大家有兴趣，可以从网上获取更多与 CGroups 相关的资料，这里就不展开解读了。

### 16.2.3 容器与控制组

在容器技术中使用 CGroups，关键是利用它可度量、可配额的特性。如图 16-2 所示，由于 Docker 使用了 Linux Container 技术，所以可以通过 CGroups 对容器使用的资源进行控制。

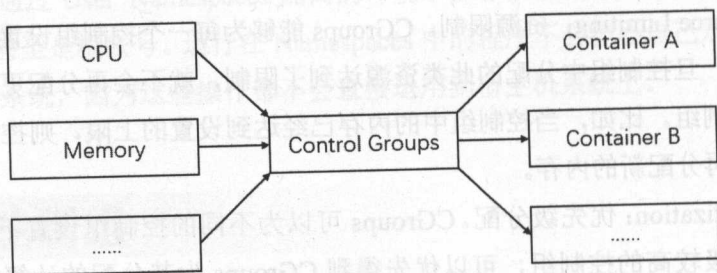


图 16-2 容器与控制组的关系

CGroups 主要能够从 blkio、cpu、cpuacct、cpuset、devices、freezer、memory、net\_cls 和 ns 等方面限制计算机资源的使用。每个方面都代表一个 CGroups 子系统，它们各自的含义分别如下。

- ❑ blkio: 块设备 IO 控制。这个子系统设置限制每个块设备的输入输出控制，如磁盘、光盘及 USB 等。
- ❑ cpu: CPU 使用控制。

- ❑ `cpuacc`: CPU 资源报告。
- ❑ `cpuset`: 多核 CPU 核心使用控制。
- ❑ `devices`: 设备访问控制。
- ❑ `freezer`: 进程挂起控制。
- ❑ `memory`: 内存使用控制。
- ❑ `net_cls`: 网络控制。
- ❑ `ns`: 名称空间子系统。

通过 `docker create` 或 `docker run` 命令创建容器时，可以通过附加的参数配置对资源的使用，见表 16-1。

表 16-1 控制资源使用的参数

<code>--blkio-weight</code>	块设备IO使用权重设置，参照于Docker本身，值为10~1000
<code>--blkio-weight-device</code>	块设备IO使用权重设置，参照于宿主机
<code>--cpu-percent</code>	CPU使用率限制
<code>--cpu-period</code>	限制CPU时间配额的时间，需要与 <code>--cpu-quota</code> 一起使用
<code>--cpu-quota</code>	限制CPU时间配额的配额，需要与 <code>--cpu-period</code> 一起使用
<code>-c, --cpu-shares</code>	CPU使用权重设置
<code>--cpuset-cpus</code>	设置使用哪个CPU核心
<code>--cpuset-mems</code>	设置使用哪块CPU缓存
<code>--device-read-bps</code>	限制设备读取速度，单位为字节每秒
<code>--device-read-iops</code>	限制设备读取速度，单位为IO数每秒
<code>--device-write-bps</code>	限制设备写入速度，单位为字节每秒
<code>--device-write-iops</code>	限制设备写入速度，单位为IO数每秒
<code>--io-maxbandwidth</code>	限制设备带宽
<code>--io-maxiops</code>	限制设备每秒IO数
<code>--kernel-memory</code>	核心内存使用限制
<code>-m, --memory</code>	内存使用量限制
<code>--memory-reservation</code>	保留内存限制

--memory-swap	交换区内内存使用量限制
--memory-swappiness	交换区内内存占比限制

## 16.3 联合文件系统

UFS (Union File System) 是联合文件系统的简称。所谓联合文件系统, 就是能够把多个目录挂载成同一个目录的文件系统。这种挂载方式的特点是不但能够从不同的文件系统里把文件挂载到虚拟的联合文件系统中, 还能够把多个目录挂载合并成同一个目录。

### 16.3.1 关于 UFS

在 Linux 的各种发行版本中, 有一个专门为企业、学校、政府部门设计地用于演示的系统——Knoppix。它的主要功能是读取 CD、DVD 中的内容, 并把它们展示给用户。

CD、DVD 等光盘有一个很大的缺陷, 就是一旦写入就无法修改。但在实际使用的过程中, 我们难免需要对其中的数据进行一些修改。图 16-3 展示了 Knoppix 的标识。此时, 可以将 CD、DVD 中的数据复制到硬盘, 再对硬盘中的数据进行修改, 但是用于演示的 Knoppix 系统往往是搭建在小型机器上的。在这些机器上加装硬盘, 显然不是好的解决方法。于是 Knoppix 采用了一种特殊的方式来解决这个问题, 即只记录文件的修改, 而不把所有的数据存储到可读写存储器中。这样就可以通过容量较小的 U 盘来存储数据的更改, 而不需要依赖硬盘。

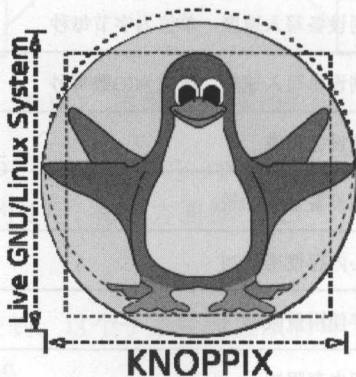


图 16-3 Knoppix 的标识



联合文件系统的作用，我们从上面的场景中就能够看出来。联合文件系统将多个目录挂载成同一个目录，其中的修改只作用于最顶层的目录上，而下面的目录只作为可读层。对于光盘和 U 盘两个文件系统，我们可以把光盘挂载到联合文件系统的下方，把 U 盘挂载到联合文件系统的上方，这样通过软件对联合文件系统进行读取时，就能读取到光盘中的数据，修改的内容也会作用到 U 盘的文件系统中。

### 16.3.2 Docker 中的 UFS

在 Docker 中，采用的文件系统是 AUFS (Another Union File System)，这是一个 UFS 的实现，兼容了 UFS 的功能，并在其基础上搭建了更完善的体系。

Docker 不但通过 AUFS 实现了从不同文件系统中挂载目录的目的，还通过 UFS 特殊的设计实现了 Docker 镜像的基础。图 16-4 展示了 UFS 实现的容器与系统的关系。

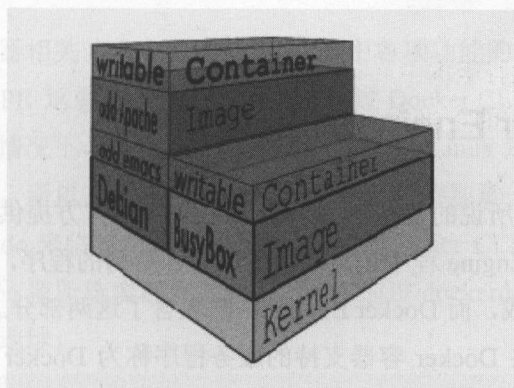


图 16-4 Docker 中的 UFS

Docker 镜像及容器中所使用的沙盒环境中的文件系统，类似于 Git 提交的形式，每一个镜像层都可以看成一次文件的提交。而实现镜像层技术的基础，正是来自联合文件系统。Docker 为正在修改的层挂载了一个可读写的目录结构，而对于下层镜像，Docker 只以只读形式挂载，因为它们只提供文件读取的功能。

Docker 中的每个镜像层都有三种不同的权限分配形式。

- ☐ rw: Read-Write, 可写可读。
- ☐ ro: Read-Only, 只读。
- ☐ rr: Real-Read-Only, 在只读基础上，不接收任何文件改动的信号，提高了效率。

Docker 利用 UFS 实现的镜像技术，是 Docker 容器技术能够被广泛使用的一个杀手锏。相较于以往的虚拟机镜像，UFS 制成的镜像大幅缩小了占用空间，并且能够以分拆的方式在镜像之间共享数据。特别是对于镜像的更新，Docker 能够在不修改原有镜像的同时，以最小的空间消耗来存储对文件的修改。

当然，Docker 在逐渐发展的过程中，也纳入了更多的 UFS 实现，如 btrfs、vfs、DeviceMapper 等。我们可以通过 `dockerd` 命令，以及 `-s` 或 `--storage-driver` 参数，来修改 Docker 使用的存储驱动。

## 16.4 Docker Engine 架构

Docker Engine 是 Docker 中的核心程序，包含了 Docker 核心的服务部件——Docker Daemon。

### 16.4.1 Docker Engine 的组成结构

通常情况下，我们所说的 Docker 指的是由 Docker 官方提供的 Docker Engine。如图 16-5 所示，Docker Engine 程序是一个标准的 C/S 结构的程序，即由客户端（Client）与服务端（Server）组成，而 Docker Engine 同时包含了这两部分。通常，我们把 Docker Engine 中所包含的提供 Docker 容器支持的服务程序称为 Docker Daemon，把能够操作 Docker Engine 的客户端程序称为 Docker CLI。

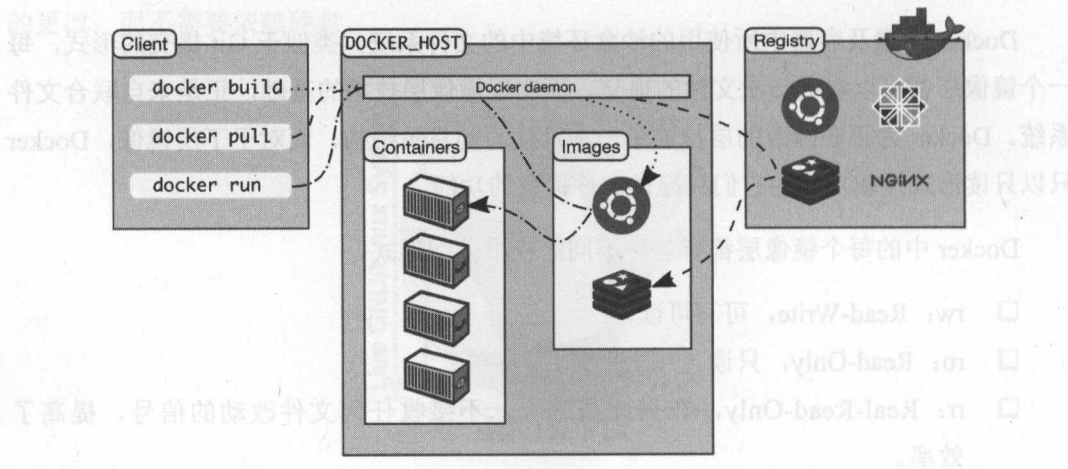


图 16-5 Docker Engine 体系结构

Docker Daemon 与 Docker CLI 之间是通过 Docker API 进行通信交流的，也就是说，通过 Docker CLI 下达的命令，通过 Docker CLI 转换为对应的 Docker API 后，被提交给了 Docker Daemon。所以，虽然 Docker Engine 同时包含了 Docker 服务端 Docker Daemon 和客户端 Docker CLI，但两者是非常松散、不相互依存的。Docker CLI 既可以直接通过进程间通信的方式与本机的 Docker Daemon 连接，也能通过 socket 向运行在其他机器上的 Docker Daemon 下达操作指令，甚至能够同时操作多个 Docker Daemon。

## 16.4.2 Docker Daemon

Docker Daemon 是实现 Docker 核心功能的程序，管理包括镜像、容器、网络、数据卷等在内的所有 Docker 组成模块。只有 Docker Daemon 运行时，我们才能使用到它所提供的容器技术支持，所以 Docker Daemon 通常以服务的形式常驻宿主机中，而我们之前谈到的启动 Docker 服务，其实指的就是启动 Docker Daemon。

除了管理所有与容器相关的内容，维持 Docker 中各项功能的运转，Docker Daemon 还对外提供了 Docker API，这就是为什么我们能够通过 Docker CLI 对 Docker Daemon 进行操作的原因。在默认情况下，Docker Daemon 使用的是 Linux 进程间通信端口，且只允许本地的根用户访问。所以在没有使用根用户登录时，特别是一些系统没有提供根用户的时候，需要通过 `sudo` 来临时切换到根用户，执行 Docker CLI 提供的 `docker` 命令。如果想让 Docker Daemon 监听选定的网络地址，可以通过 `dockerd` 命令，配合 `-H` 或 `--host` 参数，给出监听的网络地址。

```
$ sudo dockerd -H unix:///var/run/docker.sock -H tcp://0.0.0.0:9876
```

除了设置 Docker Daemon 的监听端口，`dockerd` 命令还能对 Docker Daemon 进行更多的配置，如之前谈到的 Docker 文件存储驱动方案、DNS 服务器、默认网桥、镜像仓库的镜像服务器、日志文件的存放路径等。关于这些参数的使用方法，我们可以通过 `dockerd` 命令查看到。

```
$ sudo dockerd --help
```

修改 Docker Daemon 配置，可以通过 `dockerd` 命令实现，也可以通过修改 Docker 的配置文件来实现。通过 Docker 的配置文件来设置 Docker，配置能够持久保存在文件中，且清晰易维护。

在 Linux 系统下，Docker 的配置文件位于 `/etc/docker/daemon.json` 中，其内容以 JSON 形成书写，通常的格式如下。



```
{
  "authorization-plugins": [],
  "dns": [],
  "dns-opts": [],
  "dns-search": [],
  "exec-opts": [],
  "exec-root": "",
  "storage-driver": "",
  "storage-opts": [],
  "labels": [],
  "live-restore": true,
  "log-driver": "",
  "log-opts": {},
  "mtu": 0,
  "pidfile": "",
  "graph": "",
  "cluster-store": "",
  "cluster-store-opts": {},
  "cluster-advertise": "",
  "max-concurrent-downloads": 3,
  "max-concurrent-uploads": 5,
  "debug": true,
  "hosts": [],
  "log-level": "",
  "tls": true,
  "tlsverify": true,
  "tlscacert": "",
  "tlscert": "",
  "tlskey": "",
  "swarm-default-advertise-addr": "",
  "api-cors-header": "",
  "selinux-enabled": false,
  "userns-remap": "",
  "group": "",
  "cgroup-parent": "",
  "default-ulimits": {},
  "ipv6": false,
  "iptables": false,
  "ip-forward": false,
  "ip-masq": false,
  "userland-proxy": false,
  "ip": "0.0.0.0",
```

```

"bridge": "",
"bip": "",
"fixed-cidr": "",
"fixed-cidr-v6": "",
"default-gateway": "",
"default-gateway-v6": "",
"icc": false,
"raw-logs": false,
"registry-mirrors": [],
"insecure-registries": [],
"disable-legacy-registry": false,
"default-runtime": "runc",
"oom-score-adjust": -500,
"runtimes": {
  "runc": {
    "path": "runc"
  },
  "custom": {
    "path": "/usr/local/bin/my-runc-replacement",
    "runtimeArgs": [
      "--debug"
    ]
  }
}
}

```

在启动 Docker Daemon 时，还可以通过--config-file 参数指定其他路径中的文件作为配置文件。

在 Windows 系统中，Docker 的配置文件在%programdata%\docker\config\daemon.json 文件中，其内容与 Linux 系统下的配置有所不同，大致的条目如下。

```

{
  "authorization-plugins": [],
  "dns": [],
  "dns-opts": [],
  "dns-search": [],
  "exec-opts": [],
  "storage-driver": "",
  "storage-opts": [],
  "labels": [],
  "live-restore": true,
  "log-driver": "",

```



```
"mtu": 0,  
"pidfile": "",  
"graph": "",  
"cluster-store": "",  
"cluster-advertise": "",  
"debug": true,  
"hosts": [],  
"log-level": "",  
"tlsverify": true,  
"tlscacert": "",  
"tlscert": "",  
"tlskey": "",  
"swarm-default-advertise-addr": "",  
"group": "",  
"default-ulimits": {},  
"bridge": "",  
"fixed-cidr": "",  
"raw-logs": false,  
"registry-mirrors": [],  
"insecure-registries": [],  
"disable-legacy-registry": false  
}
```

在 Windows 系统里，我们更推荐使用 Docker for Windows 去配置 Docker。其图像界面和清晰的配置说明，能够让我们更快、更准确地修改 Docker 的配置。

### 16.4.3 Docker CLI

Docker CLI 其实就是在终端命令行中使用的 docker 程序，能够将下达的命令行指令转换为 Docker API 请求，再发送到与之关联的 Docker Daemon 中。图 16-6 展示了 Docker 指令的处理流程。

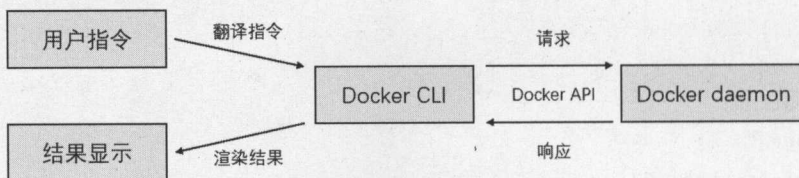


图 16-6 Docker 指令的处理流程

每个输入到终端中的 docker 指令，如果与操作 Docker Daemon 相关，则先由 Docker



CLI 根据指令的类型转换为对应的 Docker API。之后根据附加在 docker 命令后的参数，填充 Docker API 的请求参数。然后 Docker CLI 会请求与之关联的 Docker Daemon 并等待处理结果。Docker Daemon 收到 Docker CLI 发出的操作指令后，会根据指令的内容进行相应的操作，并把操作结果返回给 Docker CLI。Docker CLI 收到 Docker Daemon 返回的结果后，会解析结果中的数据，并渲染成命令行显示数据的合适方式，再打印到终端中。

在默认情况下，Docker CLI 会通过本地的进程间通信协议连接本机的 Docker Daemon。我们可以通过显示指定管理 Docker Daemon 的方式，修改 Docker CLI 操作的 Docker Daemon，只要在 docker 命令中传入 -H 或 --host 参数设置即可。

```
$ sudo docker -H tcp://127.0.0.1:9876 version
```

通常，Docker CLI 操作的就是本机中的 Docker Daemon。当然，在使用 Docker Machine、Docker Swarm 这类工具时，它们能够通过网络控制非本机的 Docker Daemon。例如，我们在 Windows 中运行 Docker for Windows，并通过命令行程序 PowerShell 输入 docker version 命令，查看 Docker CLI 和 Docker Daemon 的版本。

```
docker version
Client:
 Version:      1.12.1
 API version:  1.24
 Go version:   gol.6.3
 Git commit:   23cf638
 Built:        Thu Aug 18 17:52:38 2016
 OS/Arch:     windows/amd64

Server:
 Version:      1.12.1
 API version:  1.24
 Go version:   gol.6.3
 Git commit:   23cf638
 Built:        Thu Aug 18 17:52:38 2016
 OS/Arch:     linux/amd64
```

由于 Docker 是基于 Linux Container 技术的，目前仍然需要运行在 Linux 系统中，所以在 Windows 中使用的 Docker，其实是运行在虚拟机中的 Linux 系统所提供的。我们看到，Docker CLI 程序所显示的系统为 Windows，也就是我们所使用的系统，而 Docker Daemon 所在的系统则是虚拟机中的 Linux 系统。

## 16.5 本章小结

在本章中，我们讲解了为 Docker 提供容器技术支持的 Linux Container 中的命名空间（Namespaces）和控制组（Cgroups），也介绍了 Docker 镜像和容器沙盒运行环境所使用的联合文件系统（Union File System），还对 Docker Engine 的组织框架进行了拆解和说明。

通过本章的介绍，我们对实现 Docker 的底层技术有了一定的认知。掌握 Docker 的底层架构，有利于更好地组合 Docker 的功能，更快、更准确地实现我们的需求。对于在使用 Docker 时遇到的问题，在了解 Docker 的实现原理后，我们也能及时发现和解决。



微 信



新浪微博

敬请关注微信公众平台和  
新浪微博二维码

Docker 作为高效的容器虚拟化解决方案，正在以令人惊叹的速度快速发展。不论是在软件部署领域还是在程序开发领域，都已经且越来越多地能够看到 Docker 的身影。作为与技术相关的任何岗位的人员，都可以使用 Docker 来提高工作效率，而使用 Docker 自然也就成为了一项必不可少的技能。在日新月异的变化之中，想要学习和掌握好 Docker，《没什么难的 Docker 入门与开发实战》会是你 Docker 之路上不可或缺的灯塔。

---





Docker 技术屏蔽了开发平台的差异,简化了开发环境配置,让互联网应用开发、测试、维护更加高效快捷,有力地降低了应用开发难度,让开发效率更加高效。Docker 技术让应用部署既可以满足快速部署和资源调度所需的性能速度,又能够满足资源直接的隔离需求,为 ICT 产业的发展带来新一轮的技术变革。

本书作者对 Docker 技术具有多年的研究和实践经验,在写作过程中收集参考了大量的全新资料。本书从不同维度由浅入深地解读了 Docker 的概念、原理、使用方法、实践案例以及周边工具的使用,对 Docker 技术初学者来说是非常有帮助的入门指导书籍,对专业 Docker 开发人员而言也是非常有用的参考手册。

—— 华为南京研究所 袁博

程序员经常抱怨的一句话就是“在我自己的电脑上明明没问题啊?”更严重的情况就是,在本地开发环境运行正常的项目,部署到线上生产环境后突然出现各种 bug 和异常,而 Docker 的出现解决了这些问题。本书深入浅出地讲解了 Docker 的方方面面,不论你是还未使用 Docker 的编程初学者,还是已经使用 Docker 的资深工程师,本书都值得你认真阅读。

—— Flarum 中文社区创始人 justjavac

Docker 作为一个为开发者与系统管理员提供发布和运行分布式应用程序的平台,在轻量、便捷、高性能上的优势表现得十分明显,大大简化了开发者与系统管理员在应用程序组件组装、测试环境生产环境部署构建中的复杂程度,提高了部署效率。

本书非常详尽地为读者讲解了 Docker 在实际生产环境中的应用,通过作者的切身经验,为读者创造了提供获取 Docker 的第一手知识的渠道。了解学习 Docker,从本书开始!

—— 搜留信息科技 CTO 阿晖

Docker 是 Github 上经典的项目之一,Docker 的出现极大地提升了软件交付的能力。目前,各大厂商都在倡导微服务化架构,而 Docker 为微服务化架构的实现提供了技术基础。越来越多的公司开始考虑使用 Docker 的方式部署和管理应用,以降低运维的成本。在今天,作为开发者如果还不知道 Docker 的话,快看看这本书吧。这本书从 Docker 的基础概念说起,由浅入深,引人入胜,是学习 Docker 的好帮手。

—— 不求闻达于诸侯的全栈工程师 reeco



责任编辑：李 冰

封面设计：朝天世纪

上架建议 计算机——程序设计

ISBN 978-7-121-31427-8



定价：69.00元